Arm® A32/T32 Instruction Set

for A-profile architecture

DDI 0597 (ID030323)

Arm A32/T32 Instruction Set
for A-profile architecture

Release Information

For information on the change history and known issues for this release, see the **Release Notes** in the **A32/T32 ISA XML for A-profile architecture (2023-03)**.

Proprietary Notice

Arm Limited. Company 02557590 registered in England.
110 Fulbourn Road, Cambridge, England CB1 9NJ.

(LES-PRE-20349 version 21.0)

Confidentiality Status

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

Product Status

The information in this release covers multiple versions of the architecture. The content relating to different versions is given different quality ratings.

The information relating to the 2022 Extensions of the A-profile Architecture is at Alpha quality. Alpha quality means that most major features of the specification are described in this release, but some features and details might be missing.

The information relating to the rest of the Architecture is at Beta quality. Beta quality means that all major features of the specification are described, but some details might be missing.

Web Address

`http://www.arm.com`

Progressive Terminology Commitment

Arm values inclusive communities. Arm recognizes that we and our industry have used terms that can be offensive. Arm strives to lead the industry and create change.

Previous issues of this document included terms that can be offensive. We have replaced these terms. If you find offensive terms in this document, please contact `terms@arm.com`.

AArch32 -- Base Instructions (alphabetic order)

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.

QDSUB: Saturating Double and Subtract.

QSAX: 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).

ROR (register): Rotate Right (register): an alias of MOV, MOVS (register-shifted register).

RORS (immediate): Rotate Right, setting flags (immediate): an alias of MOV, MOVS (register).

RORS (register): Rotate Right, setting flags (register): an alias of MOV, MOVS (register-shifted 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).

RSB, RSBS (register-shifted register): Reverse Subtract (register-shifted 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, SBCS (immediate): Subtract with Carry (immediate).

SBC, SBCS (register): Subtract with Carry (register).

SBC, SBCS (register-shifted register): Subtract with Carry (register-shifted register).

SBFX: Signed Bit Field Extract.

SDIV: Signed Divide.

SEL: Select Bytes.

SETEND: Set Endianness.

SETPAN: 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.

SMLABB, SMLABT, 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.

SMLSLD, SMLSLDX: 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.

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.

UADD8: 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.

UHSAX: 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_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11–0
!= 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 = UInt(Rd);  n = UInt(Rn);  setflags = (S == '1');  imm32 = A32ExpandImm(imm12);
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3–0	15	14–12	11–8	7–0
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 = 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, 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 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>.

<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, 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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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>{<q>} {<Rdn>,} <Rdn>, <Rm> // (Inside IT block)
```

```
ADCS{<q>} {<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

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	1	0	1	0	S	Rn				(0)	imm3			Rd				imm2		stype		Rm			

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”:

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 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<c> <Rd>, <Rn>, <Rd> has <Rd> and <Rn> both in the range R0-R7, it is assembled using encoding T1 as though ADC<c> <Rd>, <Rn> had been written.

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

Operation

```
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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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.

A1

31 30 29 28	27	26	25	24	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	Rs	0	stype	1	Rm
cond															

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

<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 = 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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

It has encodings from the following instruction sets: A32 (A1) and T32 (T1 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	1	0	0	0	1	1	1	1	Rd	imm12
cond														

A1

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

is equivalent to

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

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

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

is equivalent to

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

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	0	i	1	0	0	0	0	0	1	1	1	1	0	imm3	Rd	imm8

T3

```
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 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 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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11–0
!= 1111	0	0	1	0	1	0	0	S	Rn	Rd	imm12
cond											

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

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

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

```
ADDS{<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	8–6	5–3	2–0
0	0	0	1	1	1	0	imm3	Rn	Rd

T1

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

ADDS{<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	10–8	7–0
0	0	1	1	0	Rdn	imm8

T2

```
ADD<c>{<q>} <Rdn>, #<imm8> // (Inside IT block, and <Rdn>, <imm8> can be represented in T1)
```

```
ADD<c>{<q>} {<Rdn>,} <Rdn>, #<imm8> // (Inside IT block, and <Rdn>, <imm8> cannot be represented in T1)
```

```
ADDS{<q>} <Rdn>, #<imm8> // (Outside IT block, and <Rdn>, <imm8> can be represented in T1)
```

```
ADDS{<q>} {<Rdn>,} <Rdn>, #<imm8> // (Outside IT block, and <Rdn>, <imm8> cannot be represented in T1)
```

```
d = UInt(Rdn);  n = UInt(Rdn);  setflags = !InITBlock();  imm32 = ZeroExtend(imm8, 32);
```

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	0	i	0	1	0	0	0	S	!= 1101	0	imm3	Rd	imm8
												Rn				

ADD (S == 0)

```
ADD<c>.W {<Rd>,} <Rn>, #<const> // (Inside IT block, and <Rd>, <Rn>, <const> can be represented in T1 or
```

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

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

```
ADDS.W {<Rd>,} <Rn>, #<const> // (Outside IT block, and <Rd>, <Rn>, <const> can be represented in T1 or T
```

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

```
if Rd == '1111' && S == '1' then SEE "CMN (immediate)";
if Rn == '1101' then SEE "ADD (SP plus immediate)";
d = UInt(Rd);  n = UInt(Rn);  setflags = (S == '1');  imm32 = T32ExpandImm(i:imm3:imm8);
if (d == 15 && !setflags) || n == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

T4

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	1	0	0	0	0	0	!= 11x1	0	imm3	Rd	imm8
												Rn				

T4

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

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

```
if Rn == '1111' then SEE "ADR";
if Rn == '1101' then SEE "ADD (SP plus immediate)";
d = UInt(Rd);  n = UInt(Rn);  setflags = FALSE;  imm32 = ZeroExtend(i:imm3:imm8, 32);
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.
<imm8>	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 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, T3 and T4: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>.

<Rn> For encoding A1 and T4: is the general-purpose source register, encoded in the "Rn" field. If the SP is used, see *ADD (SP plus immediate)*. If the PC is used, see *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 *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

```
if CurrentInstrSet() == InstrSet_A32 then
    if ConditionPassed() then
        EncodingSpecificOperations();
        (result, nzcv) = AddWithCarry(R[n], 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;
else
    if ConditionPassed() then
        EncodingSpecificOperations();
        (result, nzcv) = AddWithCarry(R[n], imm32, '0');
        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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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	S	!= 1101	Rd	imm5	stype	0	Rm
cond									Rn					

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	Rm	Rn	Rd

T1

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

```
ADDS{<q>} {<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

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		
									Rm						

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

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

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

```
if (DN:Rdn) == '1101' || Rm == '1101' then SEE "ADD (SP plus register)";
d = UInt(DN:Rdn);  n = d;  m = UInt(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

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	1	0	0	0	S	!= 1101				(0)	imm3			Rd				imm2		stype		Rm			
												Rn																			

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> // (<Rd> == <Rn>, and <Rd>, <Rn>, <Rm> can be represented in T2)
```

```
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 = 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, 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 "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”:

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 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

```
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);
    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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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

31 30 29 28	27	26	25	24	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	S	Rn	Rd	Rs	0	stype	1	Rm
cond															

Flag setting (S == 1)

```
ADDS{<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

<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 = 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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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

31-28	27	26	25	24	23	22	21	20	19	18	17	16	15-12	11-0
!= 1111	0	0	1	0	1	0	0	S	1	1	0	1	Rd	imm12
cond														

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	10-8	7-0
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	6-0
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

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	0	0	S	1	1	0	1	0	imm3			Rd				imm8							

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

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	1	0	0	0	0	0	1	1	0	1	0	imm3			Rd				imm8							

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(R[13], 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;
```

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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

31 30 29 28	27 26 25 24	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	S	1 1 0 1	Rd	imm5	stype	0	Rm
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

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

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;
```

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	1	!= 1101				1	0	1
									Rm						

T2

```
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);
```

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	1	1	0	0	0	S	1	1	0	1	(0)	imm3			Rd				imm2		stype		Rm			

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>}
```

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

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

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

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

```
if Rd == '1111' && S == '1' then SEE "CMN (register)";
d = UInt(Rd);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(stype, imm3:imm2);
if (d == 15 && !setflags) || 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*.

SP, — Is the stack pointer.

<Rdm> — Is the general-purpose destination and second source register, encoded in the "Rdm" field. If omitted, this register is the SP. 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 a simple branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC*.

<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 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”:

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 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[13], 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_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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

31-28	27	26	25	24	23	22	21	20	19	18	17	16	15-12	11-0
!= 1111	0	0	1	0	1	0	0	0	1	1	1	1	Rd	imm12
cond														

A1

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

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

A2

31-28	27	26	25	24	23	22	21	20	19	18	17	16	15-12	11-0
!= 1111	0	0	1	0	0	1	0	0	1	1	1	1	Rd	imm12
cond														

A2

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

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

T1

15	14	13	12	11	10-8	7-0
1	0	1	0	0	Rd	imm8

T1

```
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	15	14-12	11-8	7-0
1	1	1	1	0	i	1	0	1	0	1	0	1	1	1	1	0	imm3	Rd	imm8

T2

```
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
```

T3

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

T3

```
ADR{<c>}.W <Rd>, <label> // (<Rd>, <label> can be presented in T1)
```

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

```
d = UInt(Rd);  imm32 = ZeroExtend(i:imm3:imm8, 32);  add = TRUE;
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*.

<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

Alias	Of variant	Is preferred when
ADD (immediate, to PC)		Never
SUB (immediate, from PC)	T2	i:imm3:imm8 == '000000000000'
SUB (immediate, from PC)	A2	imm12 == '000000000000'

Operation

```
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

31 30 29 28	27	26	25	24	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	0	0	S	Rn	Rd	imm12
cond											

AND (S == 0)

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

ANDS (S == 1)

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

```
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	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	S	Rn	0	imm3	Rd	imm8

AND (S == 0)

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

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

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

```
if Rd == '1111' && S == '1' then SEE "TST (immediate)";
d = UInt(Rd);  n = UInt(Rn);  setflags = (S == '1');
(imm32, carry) = T32ExpandImm_C(i:imm3:imm8, PSTATE.C);
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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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

31 30 29 28	27 26 25 24	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	S	Rn	Rd	imm5	stype	0	Rm
cond									

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

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

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

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

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

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

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

```
ANDS{<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 0	Rm	Rdn

T1

```
AND<c>{<q>} {<Rdn>,} <Rdn>, <Rm> // (Inside IT block)
```

```
ANDS{<q>} {<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

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	0	0	S	Rn				(0)	imm3			Rd				imm2		stype		Rm			

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

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

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

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

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

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

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

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

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

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

```
if Rd == '1111' && S == '1' then SEE "TST (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, 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 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 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”:

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 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

```
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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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.

A1

31 30 29 28	27 26 25 24	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	S	Rn	Rd	Rs	0	stype	1	Rm
cond										

Flag setting (S == 1)

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

Not flag setting (S == 0)

```
AND{<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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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, any CONSTRAINED UNPREDICTABLE behavior, and any operational information 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)	(0)	Rd				imm5					1	0	0	Rm			
cond											S														stype						

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

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	0	imm5					Rm			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

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	0	0	1	1	1	1	(0)	imm3			Rd				imm2		1	0	Rm			
											S															stype					

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

<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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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, any CONSTRAINED UNPREDICTABLE behavior, and any operational information 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 0 1 1	0 1	0	(0) (0) (0) (0)	Rd	Rs	0	1 0	1	Rm
cond			S					stype		

Not flag setting

```
ASR{<c>}{<q>} {<Rd>,} <Rm>, <Rs>
```

is equivalent to

```
MOV{<c>}{<q>} <Rd>, <Rm>, ASR <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 0 0	Rs	Rdm
	op		

Arithmetic shift right

```
ASR<c>{<q>} {<Rdm>,} <Rdm>, <Rs> // (Inside IT block)
```

is equivalent to

```
MOV<c>{<q>} <Rdm>, <Rdm>, ASR <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	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	0	Rm	1 1 1 1	Rd	0 0 0 0	Rs
	stype	S					

Not flag setting

```
ASR<c>.W {<Rd>,} <Rm>, <Rs> // (Inside IT block, and <Rd>, <Rm>, <shift>, <Rs> can be represented in T1)
```

```
ASR{<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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

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

A1

31-28	27	26	25	24	23	22	21	20	19	18	17	16	15-12	11-7	6	5	4	3-0
!= 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

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

and is always the preferred disassembly.

T2

15	14	13	12	11	10-6	5-3	2-0
0	0	0	1	0	imm5	Rm	Rd
			op				

T2

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

is equivalent to

```
MOVS{<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	15	14-12	11-8	7-6	5	4	3-0
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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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, any CONSTRAINED UNPREDICTABLE behavior, and any operational information 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	0	1	1	0	1	1	(0)	(0)	(0)	(0)	Rd				Rs				0	1	0	1	Rm			
cond											S														stype						

Flag setting

```
ASRS{<c>}{<q>} {<Rd>,} <Rm>, <Rs>
```

is equivalent to

```
MOVS{<c>}{<q>} <Rd>, <Rm>, ASR <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	0	0	Rs			Rdm		
						op									

Arithmetic shift right

```
ASRS{<q>} {<Rdm>,} <Rdm>, <Rs> // (Outside IT block)
```

is equivalent to

```
MOVS{<q>} <Rdm>, <Rdm>, ASR <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	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	Rm				1	1	1	1	Rd				0	0	0	0	Rs			
									stype		S																				

Flag setting

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

ASRS{<c>}{<q>} {<Rd>,} <Rm>, <Rs>
```

is equivalent to

```
MOVS{<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

31 30 29 28	27 26 25	24	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	0	imm24
cond			

A1

```
B{<c>}{<q>} <label>
```

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

T1

15 14 13 12	11 10 9 8	7 6 5 4 3 2 1 0
1 1 0 1	!= 111x	imm8
	cond	

T1

```
B<c>{<q>} <label> // (Not permitted in IT block)
```

```
if cond == '1110' then SEE "UDF";
if cond == '1111' then SEE "SVC";
imm32 = SignExtend(imm8:'0', 32);
if InITBlock() then UNPREDICTABLE;
```

T2

15 14 13 12 11	10 9 8 7 6 5 4 3 2 1 0
1 1 1 0 0	imm11

T2

```
B{<c>}{<q>} <label> // (Outside or last in IT block)
```

```
imm32 = SignExtend(imm11:'0', 32);
if InITBlock() && !LastInITBlock() 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 0	S	!= 111x	imm6	1 0	J1	0	J2	imm11
		cond						

T3

```
B<c>.W <label> // (Not permitted in IT block, and <label> can be represented in T1)
```

```
B<c>{<q>} <label> // (Not permitted in IT block)
```

```
if cond<3:1> == '111' then SEE "Related encodings";
imm32 = SignExtend(S:J2:J1:imm6:imm11:'0', 32);
if InITBlock() then UNPREDICTABLE;
```

T4

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	S	imm10										1	0	J1	1	J2	imm11										

T4

```
B{<c>}.W <label> // (<label> can be represented in T2)
```

```
B{<c>}{<q>} <label>
```

```
I1 = NOT(J1 EOR S);  I2 = NOT(J2 EOR 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);
```

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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

31 30 29 28	27	26	25	24	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	1	0	msb	Rd	lsb	0	0	1	1	1	1	1
cond																	

A1

```
BFC{<c>}{<q>} <Rd>, #<lsb>, #<width>
```

```
d = UInt(Rd);  msbit = UInt(msb);  lsbit = UInt(lsb);
if d == 15 then UNPREDICTABLE;
if msbit < lsbit then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `msbit < 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.

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	1	1	0	1	1	1	1	0	imm3	Rd	imm2	(0)	msb

T1

```
BFC{<c>}{<q>} <Rd>, #<lsb>, #<width>
```

```
d = UInt(Rd);  msbit = UInt(msb);  lsbit = UInt(imm3:imm2);
if d == 15 then UNPREDICTABLE;  // Armv8-A removes UNPREDICTABLE for R13
if msbit < lsbit then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `msbit < 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.

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.
<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.

<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 ConditionPassed() then
    EncodingSpecificOperations();
    R[d]<msbit:lsbit> = Replicate('0', (msbit-lsbit)+1);
    // Other bits of R[d] are 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_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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) .

A1

31 30 29 28	27 26 25 24 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 1 0	msb	Rd	lsb	0 0 1	!= 1111
cond						Rn

A1

```
BFI{<c>}{<q>} <Rd>, <Rn>, #<lsb>, #<width>
```

```
if Rn == '1111' then SEE "BFC";
d = UInt(Rd);  n = UInt(Rn);  msbit = UInt(msb);  lsbit = UInt(lsb);
if d == 15 then UNPREDICTABLE;
if msbit < lsbit then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `msbit < 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.

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 1	1	0	!= 1111	0	imm3	Rd	imm2	(0)	msb
						Rn						

T1

```
BFI{<c>}{<q>} <Rd>, <Rn>, #<lsb>, #<width>
```

```
if Rn == '1111' then SEE "BFC";
d = UInt(Rd);  n = UInt(Rn);  msbit = UInt(msb);  lsbit = UInt(imm3:imm2);
if d == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
if msbit < lsbit then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `msbit < 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.

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 ConditionPassed() then
    EncodingSpecificOperations();
    R[d]<msbit:lsbit> = R[n]<(msbit-lsbit):0>;
    // Other bits of R[d] are 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_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11–0
!= 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>
```

```
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	3–0	15	14–12	11–8	7–0
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>
```

```
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*.

<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 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 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 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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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	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

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

T1

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

```
BICS{<q>} {<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

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	0	1	S	Rn				(0)	imm3			Rd				imm2		stype		Rm			

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 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”:

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.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (shifted, carry) = Shift_C(R[m], shift_t, shift_n, 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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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

31 30 29 28	27 26 25 24 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	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

<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 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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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

31 30 29 28	27 26 25 24 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	imm12	0 1 1 1	imm4
cond						

A1

```
BKPT{<q>} {#}<imm>
```

```
imm16 = imm12:imm4;
if cond != '1110' then UNPREDICTABLE;  // BKPT 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

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

T1

```
BKPT{<q>} {#}<imm>
```

```
imm16 = ZeroExtend(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*. A `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	imm24
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	imm24
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	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	1	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	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	0	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

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if CurrentInstrSet() == InstrSet_A32 then
        LR = PC - 4;
    else
        LR = PC<31:1> : '1';
    bits(32) targetAddress;
    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_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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)	(1)	(1)	0	0	1	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 InITBlock() && !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];
    bits(32) next_instr_addr;
    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);
```

BX

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

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

A1

```
BX{<c>}{<q>} <Rm>
```

```
m = UInt(Rm);
```

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	0	Rm				(0)	(0)	(0)

T1

```
BX{<c>}{<q>} <Rm>
```

```
m = UInt(Rm);
if InITBlock() && !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> 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.

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.

Note

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

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

BXJ

Branch and Exchange, previously Branch and Exchange Jazelle.

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)	(1)	(1)	(1)	(1)	(1)	(1)	(1)	(1)	0	0	1	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	0	0	1	1	1	1	0	0	Rm				1	0	(0)	0	(1)	(1)	(1)	(1)	(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

<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();
    BXWritePC(R[m], BranchType_INDIR);
```

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

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	op	0	i	1	imm5					Rn		

CBNZ (op == 1)

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

CBZ (op == 0)

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

```
n = UInt(Rn);  imm32 = ZeroExtend(i:imm5:'0', 32);  nonzero = (op == '1');
if InITBlock() then UNPREDICTABLE;
```

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

Assembler Symbols

<q> See *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
    CBWritePC(PC + imm32);
```

CLRBHB

Clear Branch History clears the branch history for the current context to the extent that branch history information created before the `CLRBHB` instruction cannot be used by code before the `CLRBHB` instruction to exploitatively control the execution of any indirect branches in code in the current context that appear in program order after the instruction.

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

A1
(FEAT_CLRBHB)

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

A1

```
CLRBHB{<c>}{<q>}
```

```
if !HaveFeatCLRBHB() then EndOfInstruction();   // Instruction executes as NOP
```

T1
(FEAT_CLRBHB)

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	0	(1)	(1)	(1)	(1)	1	0	(0)	0	(0)	0	0	0	0	0	0	1	0	1	1	0

T1

```
CLRBHB{<c>}{<q>}
```

```
if !HaveFeatCLRBHB() then EndOfInstruction();   // Instruction executes as NOP
```

Assembler Symbols

<c> See *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    Hint_CLRBHB();
```

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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	0	1	0	1	0	1	1	1	(1)	(1)	(1)	(1)	(1)	(1)	(1)	(1)	(0)	(0)	(0)	(0)	0	0	0	1	(1)	(1)	(1)	(1)

A1

```
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	0	(0)	0	(1)	(1)	(1)	(1)	0	0	1	0	(1)	(1)	(1)	(1)

T1

```
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_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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>
```

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

T1

```
CLZ{<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
```

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

```
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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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

31-28	27	26	25	24	23	22	21	20	19-16	15	14	13	12	11-0
!= 1111	0	0	1	1	0	1	1	1	Rn	(0)	(0)	(0)	(0)	imm12
cond														

A1

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

```
n = UInt(Rn);  imm32 = A32ExpandImm(imm12);
```

T1

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

T1

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

```
n = UInt(Rn);  imm32 = 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();
    (result, nzcv) = AddWithCarry(R[n], imm32, '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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

CMN (register)

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

31	30	29	28	27	26	25	24	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)	imm5					stype		0	Rm			
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	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”:

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.

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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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”:

stype	<type>
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 = 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

31–28	27	26	25	24	23	22	21	20	19–16	15	14	13	12	11–0
!= 1111	0	0	1	1	0	1	0	1	Rn	(0)	(0)	(0)	(0)	imm12
cond														

A1

```
CMP{<c>}{<q>} <Rn>, #<const>
```

```
n = UInt(Rn);  imm32 = A32ExpandImm(imm12);
```

T1

15	14	13	12	11	10–8	7–0
0	0	1	0	1	Rn	imm8

T1

```
CMP{<c>}{<q>} <Rn>, #<imm8>
```

```
n = UInt(Rn);  imm32 = ZeroExtend(imm8, 32);
```

T2

15	14	13	12	11	10	9	8	7	6	5	4	3–0	15	14–12	11	10	9	8	7–0
1	1	1	1	0	i	0	1	1	0	1	1	Rn	0	imm3	1	1	1	1	imm8

T2

```
CMP{<c>}.W <Rn>, #<const> // (<Rd>, <const> can be represented in T1)

CMP{<c>}{<q>} <Rn>, #<const>
```

```
n = UInt(Rn);  imm32 = 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 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

```
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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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

31	30	29	28	27	26	25	24	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	1	Rn				(0)	(0)	(0)	(0)	imm5					stype		0	Rm			
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 = 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	0	Rm			Rn		

T1

```
CMP{<c>}{<q>} <Rn>, <Rm> // (<Rn> and <Rm> both from R0-R7)
```

```
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
0	1	0	0	0	1	0	1	N	Rm				Rn		

T2

```
CMP{<c>}{<q>} <Rn>, <Rm> // (<Rn> and <Rm> not both from R0-R7)
```

```
n = UInt(N:Rn);  m = UInt(Rm);
(shift_t, shift_n) = (SRType_LSL, 0);
if n < 8 && m < 8 then UNPREDICTABLE;
if n == 15 || m == 15 then UNPREDICTABLE;
```

CONSTRAINED 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

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	1	1	0	1	1	Rn	(0)	imm3	1	1	1	1	imm2	stype	Rm

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)
```

```
CMP{<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 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”:

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 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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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

31 30 29 28	27	26	25	24	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	1	Rn	(0)	(0)	(0)	(0)	Rs	0	stype	1	Rm
cond																		

A1

```
CMP{<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”:

stype	<type>
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 = 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 *PSTATE*.{A, I, F} interrupt mask bits and, optionally, the *PSTATE*.M mode field, without changing any other *PSTATE* bits.

`CPS` is treated as `NOP` if executed in User mode unless it is defined as being CONSTRAINED UNPREDICTABLE elsewhere in this section.

The PE checks whether the value being written to PSTATE.M is legal. See *Illegal changes to PSTATE.M*.

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	0	0	1	0	0	0	0	imod		M	0	(0)	(0)	(0)	(0)	(0)	(0)	(0)	A	I	F	0	mode				

Change mode (imod == 00 && M == 1)

```
CPS{<q>} #<mode> // (Cannot be conditional)
```

Interrupt disable (imod == 11 && M == 0)

```
CPSID{<q>} <iflags> // (Cannot be conditional)
```

Interrupt disable and change mode (imod == 11 && M == 1)

```
CPSID{<q>} <iflags> , #<mode> // (Cannot be conditional)
```

Interrupt enable (imod == 10 && M == 0)

```
CPSIE{<q>} <iflags> // (Cannot be conditional)
```

Interrupt enable and change mode (imod == 10 && M == 1)

```
CPSIE{<q>} <iflags> , #<mode> // (Cannot be conditional)
```

```
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;
```

CONSTRAINED UNPREDICTABLE behavior

If `imod == '01'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.

If `imod == '00' && M == '0'`, 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'.

T1

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

Interrupt disable (im == 1)

```
CPSID{<q>} <iflags> // (Not permitted in IT block)
```

Interrupt enable (im == 0)

```
CPSIE{<q>} <iflags> // (Not permitted in IT block)
```

```
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 InITBlock() then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `A:I:F == '000'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.

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	0	0	1	1	1	0	1	0	(1)	(1)	(1)	(1)	1	0	(0)	0	(0)	imod		M	A	I	F	mode				

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)

```
CPSID{<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

<q> See *Standard assembler syntax fields*.

<iflags> Is a sequence of one or more of the following, specifying which interrupt mask bits are affected:

a

Sets the A bit in the instruction, causing the specified effect on *PSTATE*.A, the SError interrupt mask bit.

i

Sets the I bit in the instruction, causing the specified effect on *PSTATE*.I, the IRQ interrupt mask bit.

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.

Operation

```
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);
```

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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 an Armv8.0 implementation, this is an OPTIONAL instruction. From Armv8.1, it is mandatory for all implementations to implement this instruction.

Note

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 (A1) and T32 (T1) .

A1
(FEAT_CRC32)

31 30 29 28	27	26	25	24	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)	0	1	0	0	Rm
cond												C						

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 ! 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
(FEAT_CRC32)

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	0	1	1	0	0	Rn				1	1	1	1	Rd				1	0	sz		Rm			
											C																				

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*. A 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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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 an Armv8.0 implementation, this is an OPTIONAL instruction. From Armv8.1, it is mandatory for all implementations to implement this instruction.

Note

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 (A1) and T32 (T1) .

A1
(FEAT_CRC32)

31 30 29 28	27	26	25	24	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
(FEAT_CRC32)

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	0	1	1	0	1	Rn				1	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*. A 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.

CSDB

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.

Note

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
!= 1111				0	0	1	1	0	0	1	0	0	0	0	0	(1)	(1)	(1)	(1)	(0)	(0)	(0)	(0)	0	0	0	1	0	1	0	0
cond																															

A1

```
CSDB{<c>}{<q>}
```

```
if cond != '1110' then UNPREDICTABLE;      // CSDB 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

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	0	(1)	(1)	(1)	(1)	1	0	(0)	0	(0)	0	0	0	0	0	0	1	0	1	0	0

T1

```
CSDB{<c>}{<q>}
```

```
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.

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*.

Operation

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

    ConsumptionOfSpeculativeDataBarrier();
```

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

DBG

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

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

A1

```
DBG{<c>}{<q>} #<option>
```

```
// DBG executes as a NOP. The 'option' field is ignored
```

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	0	(1)	(1)	(1)	(1)	1	0	(0)	0	(0)	0	0	0	1	1	1	1	option			

T1

```
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

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

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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 the DCPS<n> instructions, see *DCPS*.

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	1	1	1	1	0	0	0	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1

T1

```
DCPS1
```

```
// No additional decoding required.
```

Operation

```
if !Halted() then UNDEFINED;

if EL2Enabled() && PSTATE.EL == EL0 then
    tge = if ELUsingAArch32(EL2) then HCR.TGE else HCR_EL2.TGE;
    if tge == '1' then UNDEFINED;

if PSTATE.EL != EL0 || ELUsingAArch32(EL1) then
    if PSTATE.M == M32_Monitor then SCR.NS = '0';
    if PSTATE.EL != EL2 then
        AArch32.WriteMode(M32_Svc);
        PSTATE.E = SCTLR.EE;
        if 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;

    DLR = bits(32) UNKNOWN;
    DSPSR = bits(32) UNKNOWN;
else                                        // Targeting EL1 using AArch64
    AArch64.MaybeZeroRegisterUppers();
    MaybeZeroSVEUppers(EL1);
    PSTATE.nRW = '0';
    PSTATE.SP = '1';
    PSTATE.EL = EL1;
    if HavePANExt() && SCTLR_EL1.SPAN == '0' then PSTATE.PAN = '1';
    if HaveUAOExt() 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 (HaveIESB() && SCTLR_EL1.IESB == '1' &&
          !ConstrainUnpredictableBool(Unpredictable_IESBinDebug)) then
        SynchronizeErrors();

UpdateEDSCRFields();                        // Update EDSCR PE state flags
```

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 the DCPS<n> instructions, see *DCPS*.

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	1	1	1	1	0	0	0	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0

T1

```
DCPS2
```

```
if !HaveEL(EL2) then UNDEFINED;
```

Operation

```
if !Halted() || !EL2Enabled() then UNDEFINED;

if ELUsingAArch32(EL2) then
    AArch32.WriteMode(M32_Hyp);
    PSTATE.E = HSCTLR.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() && SCTLR_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() && SCTLR_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 the DCPS<n> instructions, see *DCPS*.

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	1	1	1	1	0	0	0	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1

T1

```
DCPS3
```

```
if !HaveEL(EL3) then UNDEFINED;
```

Operation

```
if !Halted() || EDSCR.SDD == '1' then UNDEFINED;

if ELUsingAArch32(EL3) then
    from_secure = CurrentSecurityState() == SS_Secure;
    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() && EffectiveEA() == '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
```

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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

31	30	29	28	27	26	25	24	23	22	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	0	1	1	1	(1)	(1)	(1)	(1)	(1)	(1)	(1)	(1)	(0)	(0)	(0)	(0)	0	1	0	1	option			

A1

```
DMB{<c>}{<q>} {<option>}
```

```
// 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	0	(0)	0	(1)	(1)	(1)	(1)	0	1	0	1	option			

T1

```
DMB{<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 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 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.

Note

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();
    MBReqDomain domain;
    MBReqTypes types;
    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);
```

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

DSB

Data Synchronization Barrier is a memory barrier that ensures the completion of memory accesses, see *Data Synchronization Barrier (DSB)*.

An AArch32 DSB instruction does not require the completion of any AArch64 TLB maintenance instructions, regardless of the nXS qualifier, appearing in program order before the AArch32 DSB.

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	0	1	1	1	(1)	(1)	(1)	(1)	(1)	(1)	(1)	(1)	(0)	(0)	(0)	(0)	0	1	0	0	!= 0x00			
																												option			

A1

```
DSB{<c>}{<q>} {<option>}
```

```
// 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	0	(0)	0	(1)	(1)	(1)	(1)	0	1	0	0	!= 0x00			
																												option			

T1

```
DSB{<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 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.

Note

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();
    boolean nXS;
    if HaveFeatXS() then
        nXS = (PSTATE.EL IN {EL0, EL1} && !ELUsingAArch32(EL2) &&
            IsHCRXEL2Enabled() && HCRX_EL2.FnXS == '1');
    else
        nXS = FALSE;
    MBReqDomain domain;
    MBReqTypes types;
    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
            assert !(option IN {'0x00'});
            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;

    DataSynchronizationBarrier(domain, types, nXS);
```

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

31 30 29 28	27	26	25	24	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	0	1	S	Rn	Rd	imm12
cond											

EOR (S == 0)

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

EORS (S == 1)

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

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

EOR (S == 0)

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

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

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

```
if Rd == '1111' && S == '1' then SEE "TEQ (immediate)";
d = UInt(Rd);  n = UInt(Rn);  setflags = (S == '1');
(imm32, carry) = T32ExpandImm_C(i:imm3:imm8, PSTATE.C);
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 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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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 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>,} <Rdn>, <Rm> // (Inside IT block)
```

```
EORS{<q>} {<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

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	1	0	0	S	Rn				(0)	imm3			Rd				imm2		stype		Rm			

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

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

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

```
EOR<c>.W {<Rd>,} <Rn>, <Rm> // (Inside IT block, and <Rd>, <Rn>, <Rm> can be represented in T1)

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

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

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

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

```
EORS.W {<Rd>,} <Rn>, <Rm> // (Outside IT block, and <Rd>, <Rn>, <Rm> can be represented in T1)

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

```
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 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>.

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”:

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 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

```
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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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.

A1

31 30 29 28	27 26 25 24	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	Rn	Rd	Rs	0	stype	1	Rm
cond										

Flag setting (S == 1)

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

Not flag setting (S == 0)

```
EOR{<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] EOR 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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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	1	1	0	(1)	(1)	(1)	(0)
cond																															

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	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

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();
    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                                          // Perform DRPS operation in Debug state
        if PSTATE.M == M32_User then
            UNDEFINED;
        elsif PSTATE.M == M32_System then
            UNPREDICTABLE;                        // UNDEFINED or NOP
        else
            SynchronizeContext();
            DebugRestorePSR();
```

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`.

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`.

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

A1

(FEAT_RAS)

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

A1

```
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

(FEAT_RAS)

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	0	(1)	(1)	(1)	(1)	1	0	(0)	0	(0)	0	0	0	0	0	0	1	0	0	0	0

T1

```
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*.

Operation

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

    SynchronizeErrors();
    AArch32.ESBOperation();
    if PSTATE.EL IN {EL0, EL1} && EL2Enabled() then AArch32.vESBOperation();
    TakeUnmaskedSErrorInterrupts();
```

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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

31 30 29 28	27	26	25	24	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	0	imm12	0	1	1	1	imm4
cond														

A1

```
HLT{<q>} {#}<imm>
```

```
if EDSCR.HDE == '0' || !HaltingAllowed() then UNDEFINED;
if cond != '1110' then UNPREDICTABLE; // HLT 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

15	14	13	12	11	10	9	8	7	6	5 4 3 2 1 0
1	0	1	1	1	0	1	0	1	0	imm6

T1

```
HLT{<q>} {#}<imm>
```

```
if EDSCR.HDE == '0' || !HaltingAllowed() then UNDEFINED;
```

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 `HLT` 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 for assembly and disassembly only. It is ignored by the PE, but can be used by a debugger to store more information about the halting breakpoint.

For encoding T1: is a 6-bit unsigned immediate, in the range 0 to 63, encoded in the "imm6" field. This value is for assembly and disassembly only. It is ignored by the PE, but can be used by a debugger to store more information about the halting breakpoint.

Operation

```
EncodingSpecificOperations();
boolean is_async = FALSE;
Halt(DebugHalt_HaltInstruction, is_async);
```

HVC

Hypervisor Call causes a Hypervisor Call exception. For more information, see *Hypervisor Call (HVC) exception*. Software executing at EL1 can use this instruction to call the hypervisor to request a service.

The HVC instruction is UNDEFINED:

- When EL3 is implemented and using AArch64, and *SCR_EL3*.HCE is set to 0.
- In Non-secure EL1 modes when EL3 is implemented and using AArch32, and *SCR*.HCE is set to 0.
- When EL3 is not implemented and either *HCR_EL2*.HCD is set to 1 or *HCR*.HCD is set to 1.
- When EL2 is not implemented.
- In Secure state, if EL2 is not enabled in the current Security state.
- In User mode.

The HVC instruction is CONSTRAINED UNPREDICTABLE in Hyp mode when EL3 is implemented and using AArch32, and *SCR*.HCE is set to 0.

On executing an HVC instruction, the *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 *Use of the HSR*.

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	imm12	0	1	1	1	imm4
cond														

A1

```
HVC{<q>} {#}<imm16>
```

```
if cond != '1110' then UNPREDICTABLE;
imm16 = imm12:imm4;
```

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

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	1	1	1	1	1	1	0	imm4	1	0	0	0	imm12

T1

```
HVC{<q>} {#}<imm16>
```

```
imm16 = imm4:imm12;
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*. An HVC instruction must be unconditional.

<imm16> — For encoding 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 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

```
EncodingSpecificOperations();
if PSTATE.EL IN {EL0, EL3} || !EL2Enabled() then
    UNDEFINED;

bit hvc_enable;
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

31	30	29	28	27	26	25	24	23	22	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	0	1	1	1	(1)	(1)	(1)	(1)	(1)	(1)	(1)	(1)	(0)	(0)	(0)	(0)	0	1	1	0	option			

A1

```
ISB{<c>}{<q>} {<option>}
```

```
// 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	0	(0)	0	(1)	(1)	(1)	(1)	0	1	1	0	option			

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();
```

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

IT

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.

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			

T1

```
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;
```

CONSTRAINED 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]

E
NOT firstcond[0]

<z> 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:

T
firstcond[0]

E
NOT firstcond[0]

<q> See *Standard assembler syntax fields*.

<cond> 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.

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

```
EncodingSpecificOperations();
AArch32.CheckITEnabled(mask);
PSTATE.IT<7:0> = firstcond:mask;
ShouldAdvanceIT = FALSE;
```

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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

31 30 29 28	27	26	25	24	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	1	Rn	Rt	(1)	(1)	0	0	1	0	0	1	(1)	(1)	(1)	(1)
cond																						

A1

```
LDA{<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	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	(1)	(1)	(1)	(1)	1	0	1	0	(1)	(1)	(1)	(1)

T1

```
LDA{<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] = MemO[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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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	1 0	1	Rn	Rt	(1) (1)	0	0	1 0 0 1	(1) (1) (1) (1)
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	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	(1) (1) (1) (1)	1	0	0 0	(1) (1) (1) (1)

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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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

31 30 29 28	27 26 25 24 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	1	Rn	Rt	(1)	(1)	1	0	1 0 0 1	(1)	(1)	(1)	(1)
cond														

A1

```
LDAEX{<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	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	(1)	(1)	(1)	(1)	1	1	1 0	(1)	(1)	(1)	(1)

T1

```
LDAEX{<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];
    AArch32.SetExclusiveMonitors(address, 4);
    R[t] = MemO[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.

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

31 30 29 28	27 26 25 24 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	1	Rn	Rt	(1)	(1)	1	0	1 0 0 1	(1) (1) (1) (1)
cond											

A1

```
LDAEXB{<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	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	(1) (1) (1) (1)	1	1	0 0	(1) (1) (1) (1)

T1

```
LDAEXB{<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];
    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

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

A1

```
LDAEXD{<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				1	1	1	1	(1)	(1)	(1)	(1)

T1

```
LDAEXD{<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;
```

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

```
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(AccessType_GPR) then value<63:32> else value<31:0>;
    R[t2] = if BigEndian(AccessType_GPR) 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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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

31 30 29 28	27	26	25	24	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	1	1	Rn	Rt	(1)	(1)	1	0	1	0	0	1	(1)	(1)	(1)	(1)
cond																						

A1

```
LDAEXH{<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	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	(1)	(1)	(1)	(1)	1	1	0	1	(1)	(1)	(1)	(1)

T1

```
LDAEXH{<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];
    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

31 30 29 28	27	26	25	24	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	1	1	Rn	Rt	(1)	(1)	0	0	1	0	0	1	(1)	(1)	(1)	(1)
cond																						

A1

```
LDAH{<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	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	(1)	(1)	(1)	(1)	1	0	0	1	(1)	(1)	(1)	(1)

T1

```
LDAH{<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, 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.

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 *DBGDTRTXint* 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 *DBGDTRTXint* 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

31 30 29 28	27	26	25	24	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	0	W	1	!= 1111	0	1	0	1	1	1	1	0	imm8
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

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	P	U	0	W	1	!= 1111	0	1	0	1	1	1	1	0	imm8
												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

<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 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":

U	+/-
0	-
1	+

<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.
    AArch32.SysRegWriteM(cp, ThisInstr(), address);

    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_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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 *DBGDTRTXint* System register. For information about memory accesses see *Memory accesses*.

In an implementation that includes EL2, the permitted `LDC` access to *DBGDTRTXint* 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

31	30	29	28	27	26	25	24	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	0	W	1	1	1	1	1	0	1	0	1	1	1	1	0	imm8							
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

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	P	U	0	W	1	1	1	1	1	0	1	0	1	1	1	1	0	imm8							

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":

U	+/-
0	-
1	+

<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

```
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.
    AArch32.SysRegWriteM(cp, ThisInstr(), address);
```

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_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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.

The PE checks the encoding that is copied to the *CPSR* for an illegal return event. See *Illegal return events from AArch32 state*.

Load Multiple (exception return) is:

- UNDEFINED in Hyp mode.
- UNPREDICTABLE in debug state, and in User mode and System mode.

A1

31–28	27	26	25	24	23	22	21	20	19–16	15	14–0
!= 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;
```

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 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 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.

ED

Empty Descending. For this instruction, a synonym for IB.

<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_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

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if PSTATE.EL == EL2 then
        UNDEFINED;
    elsif PSTATE.M IN {M32_User,M32_System} then
        UNPREDICTABLE;                        // UNDEFINED or NOP
    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] = MemS[address,4];  address = address + 4;
        new_pc_value = MemS[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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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	register_list
cond											

A1

```
LDM{<amode>}{<c>}{<q>} <Rn>, <registers_without_pc>^
```

```
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 `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.

ED
Empty Descending. For this instruction, a synonym for IB.

<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_without_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 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] = MemS[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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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

31 30 29 28	27 26 25	24	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	1	0	W	1	Rn	register_list
cond								

A1

```
LDM{IA}{<c>}{<q>} <Rn>{!}, <registers> // (Preferred syntax)

LDMFD{<c>}{<q>} <Rn>{!}, <registers> // (Alternate syntax, Full 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 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

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

T1

```
LDM{IA}{<c>}{<q>} <Rn>{!}, <registers> // (Preferred syntax)

LDMFD{<c>}{<q>} <Rn>{!}, <registers> // (Alternate syntax, Full Descending stack)
```

```
n = UInt(Rn);  registers = '00000000':register_list;  wback = (registers<n> == '0');
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.

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	0	1	0	0	0	1	0	W	1	Rn				P	M	register_list													

T2

```
LDM{IA}{<c>}.W <Rn>{!}, <registers> // (Preferred syntax, if <Rn>, '!' and <registers> can be representec

LDMFD{<c>}.W <Rn>{!}, <registers> // (Alternate syntax, Full Descending stack, if <Rn>, '!' and <registe

LDM{IA}{<c>}{<q>} <Rn>{!}, <registers> // (Preferred syntax)

LDMFD{<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 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.
!	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

Alias	Of variant	Is preferred when
POP (multiple registers)	T2	`W == '1' && Rn == '1101' && BitCount(P:M:register_list) > 1`
POP (multiple registers)	A1	`W == '1' && Rn == '1101' && BitCount(register_list) > 1`

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n];
    for i = 0 to 14
        if registers<i> == '1' then
            R[i] = MemS[address,4];  address = address + 4;
    if registers<15> == '1' then
        LoadWritePC(MemS[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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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_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	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] = MemS[address,4];  address = address + 4;
    if registers<15> == '1' then
        LoadWritePC(MemS[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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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_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)*.

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										

A1

```
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	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

T1

```
LDMDB{<c>}{<q>} <Rn>{!}, <registers> // (Preferred syntax)

LDMEA{<c>}{<q>} <Rn>{!}, <registers> // (Alternate syntax, Empty Ascending 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 `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.

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 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> 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] = MemS[address,4];  address = address + 4;
    if registers<15> == '1' then
        LoadWritePC(MemS[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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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	1	Rn	register_list
cond										

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.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n] + 4;
    for i = 0 to 14
        if registers<i> == '1' then
            R[i] = MemS[address,4];  address = address + 4;
    if registers<15> == '1' then
        LoadWritePC(MemS[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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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

31-28	27	26	25	24	23	22	21	20	19-16	15-12	11-0
!= 1111	0	1	0	P	U	0	W	1	!= 1111	Rt	imm12
cond									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 = UInt(Rt);  n = UInt(Rn);  imm32 = 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	10-6	5-3	2-0
0	1	1	0	1	imm5	Rn	Rt

T1

```
LDR{<c>}{<q>} <Rt>, [<Rn> {, #{+}<imm>}]
```

```
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm5:'00', 32);
index = TRUE;  add = TRUE;  wback = FALSE;
```

T2

15	14	13	12	11	10-8	7-0
1	0	0	1	1	Rt	imm8

T2

```
LDR{<c>}{<q>} <Rt>, [SP{, #{+}<imm>}]
```

```
t = UInt(Rt);  n = 13;  imm32 = ZeroExtend(imm8:'00', 32);
index = TRUE;  add = TRUE;  wback = FALSE;
```

T3

15	14	13	12	11	10	9	8	7	6	5	4	3-0	15-12	11-0
1	1	1	1	1	0	0	0	1	1	0	1	!= 1111	Rt	imm12
												Rn		

T3

```
LDR{<c>}.W <Rt>, [<Rn> {, #{+}<imm>}] // (<Rt>, <Rn>, <imm> can be represented in T1 or T2)
```

```
LDR{<c>}{<q>} <Rt>, [<Rn> {, #{+}<imm>}]
```

```
if Rn == '1111' then SEE "LDR (literal)";
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm12, 32); index = TRUE;  add = TRUE;
wback = FALSE; if t == 15 && InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

T4

15	14	13	12	11	10	9	8	7	6	5	4	3-0	15-12	11	10	9	8	7-0
1	1	1	1	1	0	0	0	0	1	0	1	!= 1111	Rt	1	P	U	W	imm8
												Rn						

Offset (P == 1 && U == 0 && W == 0)

```
LDR{<c>}{<q>} <Rt>, [<Rn> {, #-<imm>}]
```

Post-indexed (P == 0 && W == 1)

```
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 == '1' && U == '1' && W == '0' then SEE "LDRT";
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 (wback && n == t) || (t == 15 && InITBlock() && !LastInITBlock()) 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.

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.

For encoding T3 and 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*.

<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”:

U	+/-
0	-
1	+

\+ 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

Alias	Of variant	Is preferred when
POP (single register)	A1 (post-indexed)	`P == '0' && U == '1' && W == '0' && Rn == '1101' && imm12 == '000000000100'`
POP (single register)	T4 (post-indexed)	`Rn == '1101' && P == '0' && U == '1' && 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];
        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_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;
```

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_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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

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	0	U	1	0	1	1	1	1	1	Rt	imm12

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*.

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: 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 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*.

<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”:

U	+/-
0	-
1	+

<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

```
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

31 30 29 28	27	26	25	24	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	0	W	1	Rn	Rt	imm5	stype	0	Rm
cond														

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

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

T1

```
LDR{<c>}{<q>} <Rt>, [<Rn>, {+}<Rm>]
```

```
t = UInt(Rt);  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	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	1	!= 1111	Rt	0 0 0 0 0 0	imm2	Rm
				Rn				

T2

```
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) = (SRType_LSL, 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”:

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.

<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 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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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	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	0 0	1	!= 1111	!= 1111	imm12
			Rn	Rt	

T2

```
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	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	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
```

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 *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”:

U	+/-
0	-
1	+

\+ 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];
        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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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

31 30 29 28	27 26 25	24	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	1 1 1 1	Rt	imm12
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;
```

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 *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

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 0	U	0 0	1	1 1 1 1	!= 1111	imm12
					Rt	

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 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> 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”:

U	+/-
0	-
1	+

<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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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";
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;
```

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	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);
```

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 0 0	0 0	1	!= 1111	!= 1111	0 0 0 0 0 0	imm2	Rm
			Rn	Rt			

T2

```
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;  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

<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”:

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.

<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 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_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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

```
LDRBT{<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 `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

```
LDRBT{<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 `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

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	0	0	0	0	1	!= 1111				Rt				1	1	1	0	imm8							
												Rn																			

T1

```
LDRBT{<c>}{<q>} <Rt>, [<Rn> {, #{+}<imm>}]
```

```
if Rn == '1111' then SEE "LDRB (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> 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”:

U	+/-
0	-
1	+

For encoding A2: specifies the index register is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

U	+/-
0	-
1	+

<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
    EncodingSpecificOperations();
    if PSTATE.EL == EL2 then UNPREDICTABLE;               // Hyp mode
    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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11–8	7	6	5	4	3–0
!= 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

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	P	U	1	W	1	!= 1111	Rt	Rt2	imm8
												Rn			

Offset (P == 1 && W == 0)

```
LDRD{<c>}{<q>} <Rt>, <Rt2>, [<Rn> {, #{+/-}<imm>}]
```

Post-indexed (P == 0 && W == 1)

```
LDRD{<c>}{<q>} <Rt>, <Rt2>, [<Rn>], #{+/-}<imm>
```

Pre-indexed (P == 1 && W == 1)

```
LDRD{<c>}{<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”:

U	+/-
0	-
1	+

<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

```
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 IsAligned(address, 8) then
        data = MemA[address,8];
        if BigEndian(AccessType_GPR)  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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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

31 30 29 28	27	26	25	24	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)	U	1	(0)	0	1	1	1	1	Rt	imm4H	1	1	0	1	imm4L
cond																			

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 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'.

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 if P == 1 and W == 0.'

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	P	U	1	W	1	1	1	1	1	Rt	Rt2	imm8

T1 (!(P == 0 && W == 0))

```
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 UNKNOWN value.

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 post-indexed addressing when P == '0' and uses pre-indexed addressing otherwise. 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*.

Related encodings: *Load/Store dual, Load/Store-Exclusive, Load-Acquire/Store-Release, 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.

<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.

+/- Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

U	+/-
0	-
1	+

<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 *Use of labels in UAL instruction syntax*.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = if add then (Align(PC,4) + imm32) else (Align(PC,4) - imm32);
    if IsAligned(address, 8) then
        data = MemA[address,8];
        if BigEndian(AccessType_GPR) 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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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

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

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'.

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 first general-purpose register to be transferred, encoded in the "Rt" field. This register must be even-numbered and not R14.

<Rt2> Is the second general-purpose register to be transferred. This register must be <R(t+1)>.

<Rn> Is the general-purpose base register, encoded in the "Rn" field. The PC can be used in the offset variant.

+/- Specifies the index register is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

U	+/-
0	-
1	+

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

Operation

```
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 IsAligned(address, 8) then
        data = MemA[address,8];
        if BigEndian(AccessType_GPR) 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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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

31 30 29 28	27	26	25	24	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	1	Rn	Rt	(1)	(1)	1	1	1	0	0	1	(1)	(1)	(1)	(1)
cond																						

A1

```
LDREX{<c>}{<q>} <Rt>, [<Rn> {, {#}<imm>}]
```

```
t = UInt(Rt);  n = UInt(Rn);  imm32 = Zeros(32); // Zero offset
if t == 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	0	1	0	0	0	0	1	0	1	Rn	Rt	(1)	(1)	(1)	(1)	imm8

T1

```
LDREX{<c>}{<q>} <Rt>, [<Rn> {, #<imm>}]
```

```
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

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n] + imm32;
    AArch32.SetExclusiveMonitors(address,4);
    R[t] = 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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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 (cond)				0	0	0	1	1	1	0	1	Rn				Rt				(1)	(1)	1	1	1	0	0	1	(1)	(1)	(1)	(1)

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	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				(1)	(1)	(1)	(1)	0	1	0	0	(1)	(1)	(1)	(1)

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

31-28	27	26	25	24	23	22	21	20	19-16	15-12	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	3-0	15-12	11-8	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)	(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

```
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(AccessType_GPR) then value<63:32> else value<31:0>;
    R[t2] = if BigEndian(AccessType_GPR) 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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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
!= 1111				0	0	0	1	1	1	1	1	Rn				Rt				(1)	(1)	1	1	1	0	0	1	(1)	(1)	(1)	(1)
cond																															

A1

```
LDREXH{<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	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				(1)	(1)	(1)	(1)	0	1	0	1	(1)	(1)	(1)	(1)

T1

```
LDREXH{<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,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');
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
1 0 0 0	1	imm5	Rn	Rt

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	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	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 = 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	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	1	1	!= 1111				Rt				1	P	U	W	imm8							
												Rn																			

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 = 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*.

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”:

U	+/-
0	-
1	+

\+ 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 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

```
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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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	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";
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 *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	15 14 13 12	11 10 9 8 7 6 5 4 3 2 1 0
1 1 1 1 1 0 0 0	U	0 1	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)";
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*.

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 <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”:

U	+/-
0	-
1	+

<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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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 == 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

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

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 0 0	0 1	1	!= 1111	!= 1111	0 0 0 0 0 0	imm2	Rm
			Rn	Rt			

T2

```
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

<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”:

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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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

31 30 29 28	27	26	25	24	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	1	Rn	Rt	imm4H	1	0	1	1	imm4L
cond																

A1

```
LDRHT{<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 `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	0	1	1	Rm
cond																			

A2

```
LDRHT{<c>}{<q>} <Rt>, [<Rn>], {+/-}<Rm>
```

```
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 != 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

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	0	0	0	1	1	!= 1111				Rt				1	1	1	0	imm8							
												Rn																			

T1

```
LDRHT{<c>}{<q>} <Rt>, [<Rn> {, #{+}<imm>}]
```

```
if Rn == '1111' then SEE "LDRH (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”:

U	+/-
0	-
1	+

For encoding A2: specifies the index register is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

U	+/-
0	-
1	+

<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
    EncodingSpecificOperations();
    if PSTATE.EL == EL2 then UNPREDICTABLE;               // Hyp mode
    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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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	Rt	imm4H	1	1	0	1	imm4L
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 Rn == '1111' then SEE "LDRSB (literal)";
if P == '0' && W == '1' then SEE "LDRSBT";
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 || (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	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	0	1	!= 1111	!= 1111	imm12
												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

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	0	0	1	!= 1111	Rt	1	P	U	W	imm8
												Rn						

Offset (Rt != 1111 && P == 1 && U == 0 && W == 0)

```
LDRSB{<c>}{<q>} <Rt>, [<Rn> {, #-<imm>}]
```

Post-indexed (P == 0 && W == 1)

```
LDRSB{<c>}{<q>} <Rt>, [<Rn>], #{+/-}<imm>
```

Pre-indexed (P == 1 && W == 1)

```
LDRSB{<c>}{<q>} <Rt>, [<Rn>, #{+/-}<imm>]!
```

```
if Rt == '1111' && P == '1' && U == '0' && W == '0' then SEE "PLI";
if Rn == '1111' then SEE "LDRSB (literal)";
if P == '1' && U == '1' && W == '0' then SEE "LDRSBT";
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*.

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”:

U	+/-
0	-
1	+

\+ 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 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] + imm32) else (R[n] - imm32);
    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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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

31 30 29 28	27	26	25	24	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	Rt	imm4H	1	1	0	1	imm4L
cond																			

A1 (!(P == 0 && W == 1))

```
LDRSB{<c>}{<q>} <Rt>, <label> // (Normal form)
```

```
LDRSB{<c>}{<q>} <Rt>, [PC, #{+/-}<imm>] // (Alternative form)
```

```
if P == '0' && W == '1' then SEE "LDRSBT";
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 *LDRSB (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	15 14 13 12	11 10 9 8 7 6 5 4 3 2 1 0
1	1	1	1	1	0	0	1	U	0	0	1	1	1	1	1	!= 1111	imm12
																Rt	

T1

```
LDRSB{<c>}{<q>} <Rt>, <label> // (Preferred syntax)
```

```
LDRSB{<c>}{<q>} <Rt>, [PC, #{+/-}<imm>] // (Alternative syntax)
```

```
if Rt == '1111' then SEE "PLI";
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*.

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 <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”:

U	+/-
0	-
1	+

<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);
    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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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

31 30 29 28	27	26	25	24	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	1	0	1	Rm
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>]!
```

```
if P == '0' && W == '1' then SEE "LDRSBT";
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

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

T1

```
LDRSB{<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);
```

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 0	0 0	1	!= 1111	!= 1111	0	0	0	0	0	0	imm2	Rm
			Rn	Rt								

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 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”:

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];
    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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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	1	Rn	Rt	imm4H	1	1 0	1	imm4L
cond													

A1

```
LDRSBT{<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 `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																

A2

```
LDRSBT{<c>}{<q>} <Rt>, [<Rn>], {+/-}<Rm>
```

```
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 != 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

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	0	0	1	!= 1111	Rt	1	1	1	0	imm8
												Rn						

T1

```
LDRSBT{<c>}{<q>} <Rt>, [<Rn> {, #{+}<imm>}]
```

```
if Rn == '1111' then SEE "LDRSB (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”:

U	+/-
0	-
1	+

For encoding A2: specifies the index register is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

U	+/-
0	-
1	+

<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
    EncodingSpecificOperations();
    if PSTATE.EL == EL2 then UNPREDICTABLE;                // Hyp mode
    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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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	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(imm4H: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	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	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

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	0	1	1	!= 1111				Rt				1	P	U	W	imm8							
												Rn																			

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

<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 *LDRSH (literal)*.

+/- Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

U	+/-
0	-
1	+

\+ 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 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] + 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] = 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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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

31-28	27	26	25	24	23	22	21	20	19	18	17	16	15-12	11-8	7	6	5	4	3-0
!= 1111	0	0	0	P	U	1	W	1	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";
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

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15-12	11-0
1	1	1	1	1	0	0	1	U	0	1	1	1	1	1	1	!= 1111	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.

<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 -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”:

U	+/-
0	-
1	+

<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] = 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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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 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 <arm-defined-word>unknown</arm-defined-word>. 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	1	Rm	Rn	Rt

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);
```

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 0	0	1	1	!= 1111	!= 1111	0 0 0 0 0 0	imm2	Rm
				Rn	Rt			

T2

```
LDRSH{<c>}.W <Rt>, [<Rn>, {+}<Rm>] // (<Rt>, <Rn>, <Rm> can be represented in T1)
```

```
LDRSH{<c>}{<q>} <Rt>, [<Rn>, {+}<Rm>{, LSL #<imm>}]
```

```
if Rn == '1111' then SEE "LDRSH (literal)";
if Rt == '1111' then SEE "Related instructions";
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*.

Related instructions: *Load/store, signed (register offset)*.

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”:

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] = 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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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

31 30 29 28	27	26	25	24	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	1	Rn	Rt	imm4H	1	1	1	1	imm4L
cond																

A1

```
LDRSHT{<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 `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	1	1	Rm
cond																			

A2

```
LDRSHT{<c>}{<q>} <Rt>, [<Rn>], {+/-}<Rm>
```

```
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 != 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

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	0	1	1	!= 1111	Rt	1	1	1	0	imm8
												Rn						

T1

```
LDRSHT{<c>}{<q>} <Rt>, [<Rn> {, #{+}<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”:

U	+/-
0	-
1	+

For encoding A2: specifies the index register is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

U	+/-
0	-
1	+

<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
    EncodingSpecificOperations();
    if PSTATE.EL == EL2 then UNPREDICTABLE;               // Hyp mode
    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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

LDRT

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
!= 1111	0	1	0	0	U	0	1	1	Rn	Rt	imm12
cond											

A1

```
LDRT{<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 `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	1	1	0	U	0	1	1	Rn	Rt	imm5	stype	0	Rm
cond														

A2

```
LDRT{<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 `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

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	0	0	1	0	1	!= 1111	Rt	1	1	1	0	imm8
												Rn						

T1

```
LDRT{<c>}{<q>} <Rt>, [<Rn> {, #{+}<imm>}]
```

```
if Rn == '1111' then SEE "LDR (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> — 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”:

U	+/-
0	-
1	+

For encoding A2: specifies the index register is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

U	+/-
0	-
1	+

<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
    EncodingSpecificOperations();
    if PSTATE.EL == EL2 then UNPREDICTABLE;               // Hyp mode
    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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

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

A1

31-28	27	26	25	24	23	22	21	20	19	18	17	16	15-12	11-7	6	5	4	3-0
!= 1111	0	0	0	1	1	0	1	0	(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	10-6	5-3	2-0
0	0	0	0	0	!= 00000	Rm	Rd
			op		imm5		

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	14-12	11-8	7-6	5	4	3-0
1	1	1	0	1	0	1	0	0	1	0	0	1	1	1	1	(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_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

LSL (register)

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, any CONSTRAINED UNPREDICTABLE behavior, and any operational information 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	0	1	1	0	1	0	(0)	(0)	(0)	(0)	Rd				Rs				0	0	0	1	Rm			
cond											S														stype						

Not flag setting

```
LSL{<c>}{<q>} {<Rd>,} <Rm>, <Rs>
```

is equivalent to

```
MOV{<c>}{<q>} <Rd>, <Rm>, LSL <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	0	1	0	Rs			Rdm		
						op									

Logical shift left

```
LSL<c>{<q>} {<Rdm>,} <Rdm>, <Rs> // (Inside IT block)
```

is equivalent to

```
MOV<c>{<q>} <Rdm>, <Rdm>, LSL <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	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	0	Rm				1	1	1	1	Rd				0	0	0	0	Rs			
									stype		S																				

Not flag setting

```
LSL<c>.W {<Rd>,} <Rm>, <Rs> // (Inside IT block, and <Rd>, <Rm>, <shift>, <Rs> can be represented in T1)
```

```
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 *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, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

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

A1

31–28	27–25	24–23	22–21	20	19–16	15–12	11–7	6–5	4	3–0
!= 1111	0 0 0	1 1	0 1	1	(0) (0) (0) (0)	Rd	!= 00000	0 0	0	Rm
cond				S			imm5	stype		

MOVS, shift or rotate by value

```
LSLS{<c>}{<q>} {<Rd>,} <Rm>, #<imm>
```

is equivalent to

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

and is always the preferred disassembly.

T2

15–13	12–11	10–6	5–3	2–0
0 0 0	0 0	!= 00000	Rm	Rd
	op	imm5		

T2

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

is equivalent to

```
MOVS{<q>} <Rd>, <Rm>, LSL #<imm>
```

and is the preferred disassembly when `!InITBlock()`.

T3

15–9	8–5	4	3–0	15	14–12	11–8	7–6	5–4	3–0
1 1 1 0 1 0 1	0 0 1 0	1	1 1 1 1	(0)	imm3	Rd	imm2	0 0	Rm
		S						stype	

MOVS, shift or rotate by value

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

```
LSLS{<c>}{<q>} {<Rd>,} <Rm>, #<imm>
```

is equivalent to

```
MOVS{<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 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: 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_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

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

A1

31–28	27–23	22–21	20	19–16	15–12	11–8	7	6–5	4	3–0
!= 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

```
LSLS{<c>}{<q>} {<Rd>,} <Rm>, <Rs>
```

is equivalent to

```
MOVS{<c>}{<q>} <Rd>, <Rm>, LSL <Rs>
```

and is always the preferred disassembly.

T1

15–10	9–6	5–3	2–0
0 1 0 0 0 0	0 0 1 0	Rs	Rdm
	op		

Logical shift left

```
LSLS{<q>} {<Rdm>,} <Rdm>, <Rs> // (Outside IT block)
```

is equivalent to

```
MOVS{<q>} <Rdm>, <Rdm>, LSL <Rs>
```

and is the preferred disassembly when `!InITBlock()`.

T2

15–7	6–5	4	3–0	15–12	11–8	7–4	3–0
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

```
LSLS.W {<Rd>,} <Rm>, <Rs> // (Outside IT block, and <Rd>, <Rm>, <shift>, <Rs> can be represented in T1)

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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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, any CONSTRAINED UNPREDICTABLE behavior, and any operational information 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) (0)	Rd	imm5	0 1	0	Rm
cond			S				stype		

MOV, shift or rotate by value

```
LSR{<c>}{<q>} {<Rd>,} <Rm>, #<imm>
```

is equivalent to

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

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

is equivalent to

```
MOV<c>{<q>} <Rd>, <Rm>, LSR #<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 1	0 0 1 0	0	1 1 1 1	(0)	imm3	Rd	imm2	0 1	Rm
		S						stype	

MOV, shift or rotate by value

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

LSR{<c>}{<q>} {<Rd>,} <Rm>, #<imm>
```

is equivalent to

```
MOV{<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 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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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, 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, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

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

A1

31–28	27	26	25	24	23	22	21	20	19	18	17	16	15–12	11–8	7	6	5	4	3–0
!= 1111	0	0	0	1	1	0	1	0	(0)	(0)	(0)	(0)	Rd	Rs	0	0	1	1	Rm
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	5–3	2–0
0	1	0	0	0	0	0	0	1	1	Rs	Rdm
						op					

Logical shift right

```
LSR<c>{<q>} {<Rdm>,} <Rdm>, <Rs> // (Inside IT block)
```

is equivalent to

```
MOV<c>{<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	3–0	15	14	13	12	11–8	7	6	5	4	3–0
1	1	1	1	1	0	1	0	0	0	1	0	Rm	1	1	1	1	Rd	0	0	0	0	Rs
									stype		S											

Not flag setting

```
LSR<c>.W {<Rd>,} <Rm>, <Rs> // (Inside IT block, and <Rd>, <Rm>, <shift>, <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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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_<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, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

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

A1

31-28	27-23	22-21	20	19-16	15-12	11-7	6-5	4	3-0
!= 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

```
LSRS{<c>}{<q>} {<Rd>,} <Rm>, #<imm>
```

is equivalent to

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

and is always the preferred disassembly.

T2

15-13	12-11	10-6	5-3	2-0
0 0 0	0 1	imm5	Rm	Rd
	op			

T2

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

is equivalent to

```
MOVS{<q>} <Rd>, <Rm>, LSR #<imm>
```

and is the preferred disassembly when `!InITBlock()`.

T3

15-9	8-5	4	3-0	15	14-12	11-8	7-6	5-4	3-0
1 1 1 0 1 0 1	0 0 1 0	1	1 1 1 1	(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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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, any CONSTRAINED UNPREDICTABLE behavior, and any operational information 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 0 1 1	0 1	1	(0) (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

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

Logical shift right

```
LSRS{<q>} {<Rdm>,} <Rdm>, <Rs> // (Outside IT block)
```

is equivalent to

```
MOVS{<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	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	Rm	1 1 1 1	Rd	0 0 0 0	Rs
	stype	S					

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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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

31 30 29 28	27 26 25 24	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	CRn	Rt	1	1	1	coproc<0>	opc2	1	CRm
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

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	opc1	0	CRn	Rt	1	1	1	coproc<0>	opc2	1	CRm
					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>”:

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, 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 to7, 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(), t);
```

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

31 30 29 28	27 26 25 24 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	Rt2	Rt	1	1	1	coproc<0>	opc1	CRm
cond							coproc<3:1>					

A1

```
MCRR{<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 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	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	0	Rt2	Rt	1	1	1	coproc<0>	opc1	CRm
						coproc<3:1>					

T1

```
MCRR{<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 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>”:

coproc<0>	<coproc>
0	p14
1	p15

<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();
    AArch32.SysRegWrite64(cp, ThisInstr(), t, t2);
```

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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
cond																

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
													Ra						

T1

```
MLA{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>
```

```
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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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								

A1

```
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;
```

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	Ra	Rd	0 0	0 1	Rm

T1

```
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

```
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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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

31-28	27-23	22-21	20	19-16	15-12	11-0
!= 1111	0 0 1 1 1	0 1	S	(0) (0) (0) (0)	Rd	imm12
cond						

MOV (S == 0)

```
MOV{<c>}{<q>} <Rd>, #<const>
```

MOVS (S == 1)

```
MOVS{<c>}{<q>} <Rd>, #<const>
```

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

A2

31-28	27-23	22	21-20	19-16	15-12	11-0
!= 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 = UInt(Rd);  setflags = FALSE;  imm32 = ZeroExtend(imm4:imm12, 32);
if d == 15 then UNPREDICTABLE;
```

T1

15-13	12-11	10-8	7-0
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

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	0	0	1	0	S	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)
```

```
MOVS{<c>}{<q>} <Rd>, #<const>
```

```
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
```

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	0	i	1	0	0	1	0	0	imm4				0	imm3			Rd				imm8							

T3

```
MOV{<c>}{<q>} <Rd>, #<imm16> // (<imm16> cannot be represented in T1 or T2)
```

```
MOVW{<c>}{<q>} <Rd>, #<imm16> // (<imm16> can be represented in T1 or T2)
```

```
d = UInt(Rd);  setflags = FALSE;  imm32 = ZeroExtend(imm4:i:imm3:imm8, 32);
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: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 = imm32;
    if d == 15 then          // Can only occur for encoding A1
        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_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

MOV, MOVS (register)

Move (register) copies a value from a register 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. 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 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.

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

31–28	27	26	25	24	23	22	21	20	19	18	17	16	15–12	11–7	6–5	4	3–0
!= 1111	0	0	0	1	1	0	1	S	(0)	(0)	(0)	(0)	Rd	imm5	stype	0	Rm
cond																	

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>}
```

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

```
MOVS{<c>}{<q>} <Rd>, <Rm>, RRX
```

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

```
MOVS{<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	6–3	2–0
0	1	0	0	0	1	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<c>{<q>} <Rd>, <Rm> {, <shift> #<amount>} // (Inside IT block)
```

```
MOVS{<q>} <Rd>, <Rm> {, <shift> #<amount>} // (Outside IT block)
```

```
d = UInt(Rd);  m = UInt(Rm);  setflags = !InITBlock();
(shift_t, shift_n) = DecodeImmShift(op, 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	0	1	0	1	0	0	1	0	S	1	1	1	1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
(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> {, LSL #0} // (<Rd>, <Rm> can be represented in T1)
```

```
MOV<c>.W <Rd>, <Rm> {, <shift> #<amount>} // (Inside IT block, and <Rd>, <Rm>, <shift>, <amount> can be r
```

```
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 re
```

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

```
d = UInt(Rd);  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

<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 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: is 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: 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. The PC can be used. Arm deprecates use of the instruction if <Rd> is the PC.

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

<shift> For encoding A1 and T3: is the type of shift to be applied to the source register, encoded in “stype”:

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

For encoding T2: is the type of shift to be applied to the source register, encoded in “op”:

op	<shift>
00	LSL
01	LSR
10	ASR

<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

Alias	Of variant	Is preferred when
ASRS (immediate)	T3 (MOVS, shift or rotate by value), A1 (MOVS, shift or rotate by value)	`S == '1' && stype == '10'`
ASRS (immediate)	T2	`op == '10' && !InITBlock()`
ASR (immediate)	T3 (MOV, shift or rotate by value), A1 (MOV, shift or rotate by value)	`S == '0' && stype == '10'`
ASR (immediate)	T2	`op == '10' && InITBlock()`

Alias	Of variant	Is preferred when
LSLS (immediate)	T3 (MOVS, shift or rotate by value)	`S == '1' && imm3:Rd:imm2 != '000xxxx00' && stype == '00'`
LSLS (immediate)	A1 (MOVS, shift or rotate by value)	`S == '1' && imm5 != '00000' && stype == '00'`
LSLS (immediate)	T2	`op == '00' && imm5 != '00000' && !InITBlock()`
LSL (immediate)	T3 (MOV, shift or rotate by value)	`S == '0' && imm3:Rd:imm2 != '000xxxx00' && stype == '00'`
LSL (immediate)	A1 (MOV, shift or rotate by value)	`S == '0' && imm5 != '00000' && stype == '00'`
LSL (immediate)	T2	`op == '00' && imm5 != '00000' && InITBlock()`
LSRS (immediate)	T3 (MOVS, shift or rotate by value), A1 (MOVS, shift or rotate by value)	`S == '1' && stype == '01'`
LSRS (immediate)	T2	`op == '01' && !InITBlock()`
LSR (immediate)	T3 (MOV, shift or rotate by value), A1 (MOV, shift or rotate by value)	`S == '0' && stype == '01'`
LSR (immediate)	T2	`op == '01' && InITBlock()`
RORS (immediate)	T3 (MOVS, shift or rotate by value)	`S == '1' && imm3:Rd:imm2 != '000xxxx00' && stype == '11'`
RORS (immediate)	A1 (MOVS, shift or rotate by value)	`S == '1' && imm5 != '00000' && stype == '11'`
ROR (immediate)	T3 (MOV, shift or rotate by value)	`S == '0' && imm3:Rd:imm2 != '000xxxx00' && stype == '11'`
ROR (immediate)	A1 (MOV, shift or rotate by value)	`S == '0' && imm5 != '00000' && stype == '11'`
RRXS	T3 (MOVS, rotate right with extend)	`S == '1' && imm3 == '000' && imm2 == '00' && stype == '11'`
RRXS	A1 (MOVS, rotate right with extend)	`S == '1' && imm5 == '00000' && stype == '11'`
RRX	T3 (MOV, rotate right with extend)	`S == '0' && imm3 == '000' && imm2 == '00' && stype == '11'`
RRX	A1 (MOV, rotate right with extend)	`S == '0' && imm5 == '00000' && stype == '11'`

Operation

```
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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

MOV, MOVS (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

31 30 29 28	27	26	25	24	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)	(0)	Rd	Rs	0	stype	1	Rm
cond																		

Flag setting (S == 1)

```
MOVS{<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

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

Arithmetic shift right (op == 0100)

```
MOV<c>{<q>} <Rdm>, <Rdm>, ASR <Rs> // (Inside IT block)

MOVS{<q>} <Rdm>, <Rdm>, ASR <Rs> // (Outside IT block)
```

Logical shift left (op == 0010)

```
MOV<c>{<q>} <Rdm>, <Rdm>, LSL <Rs> // (Inside IT block)

MOVS{<q>} <Rdm>, <Rdm>, LSL <Rs> // (Outside IT block)
```

Logical shift right (op == 0011)

```
MOV<c>{<q>} <Rdm>, <Rdm>, LSR <Rs> // (Inside IT block)

MOVS{<q>} <Rdm>, <Rdm>, LSR <Rs> // (Outside IT block)
```

Rotate right (op == 0111)

```
MOV<c>{<q>} <Rdm>, <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

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	0	0	stype		S	Rm				1	1	1	1	Rd				0	0	0	0	Rs			

Flag setting (S == 1)

```
MOVS.W <Rd>, <Rm>, <shift> <Rs> // (Outside IT block, and <Rd>, <Rm>, <shift>, <Rs> can be represented in

MOVS{<c>}{<q>} <Rd>, <Rm>, <shift> <Rs>
```

Not flag setting (S == 0)

```
MOV<c>.W <Rd>, <Rm>, <shift> <Rs> // (Inside IT block, and <Rd>, <Rm>, <shift>, <Rs> can be represented i

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; // 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.

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

Alias Conditions

Alias	Of variant	Is preferred when
ASRS (register)	A1 (flag setting)	`S == '1' && stype == '10'`
ASRS (register)	T1 (arithmetic shift right)	`op == '0100' && !InITBlock()`
ASRS (register)	T2 (flag setting)	`stype == '10' && S == '1'`
ASR (register)	A1 (not flag setting)	`S == '0' && stype == '10'`
ASR (register)	T1 (arithmetic shift right)	`op == '0100' && InITBlock()`
ASR (register)	T2 (not flag setting)	`stype == '10' && S == '0'`
LSLS (register)	A1 (flag setting)	`S == '1' && stype == '00'`
LSLS (register)	T1 (logical shift left)	`op == '0010' && !InITBlock()`
LSLS (register)	T2 (flag setting)	`stype == '00' && S == '1'`
LSL (register)	A1 (not flag setting)	`S == '0' && stype == '00'`

Alias	Of variant	Is preferred when
LSL (register)	T1 (logical shift left)	`op == '0010' && InITBlock()`
LSL (register)	T2 (not flag setting)	`stype == '00' && S == '0'`
LSRS (register)	A1 (flag setting)	`S == '1' && stype == '01'`
LSRS (register)	T1 (logical shift right)	`op == '0011' && !InITBlock()`
LSRS (register)	T2 (flag setting)	`stype == '01' && S == '1'`
LSR (register)	A1 (not flag setting)	`S == '0' && stype == '01'`
LSR (register)	T1 (logical shift right)	`op == '0011' && InITBlock()`
LSR (register)	T2 (not flag setting)	`stype == '01' && S == '0'`
RORS (register)	A1 (flag setting)	`S == '1' && stype == '11'`
RORS (register)	T1 (rotate right)	`op == '0111' && !InITBlock()`
RORS (register)	T2 (flag setting)	`stype == '11' && S == '1'`
ROR (register)	A1 (not flag setting)	`S == '0' && stype == '11'`
ROR (register)	T1 (rotate right)	`op == '0111' && InITBlock()`
ROR (register)	T2 (not flag setting)	`stype == '11' && S == '0'`

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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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	imm12
cond											

A1

```
MOVT{<c>}{<q>} <Rd>, #<imm16>
```

```
d = UInt(Rd);  imm16 = imm4:imm12;
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	1	0	i	1	0	1	1	0	0	imm4	0	imm3	Rd	imm8

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

31 30 29 28	27 26 25 24	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	1	CRn	Rt	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

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	opc1	1	CRn	Rt	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 to7, 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();

    if t != 15 || AArch32.SysRegReadCanWriteAPSR(cp, ThisInstr()) then
        AArch32.SysRegRead(cp, ThisInstr(), t);
    else
        UNPREDICTABLE;
```

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

<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 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();
    AArch32.SysRegRead64(cp, ThisInstr(), t, t2);
```

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)	(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	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”:

R	<spec_reg>
0	CPSR|APSR
1	SPSR

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 = '11111000 11101111 00000011 11011111';
        psr_val = GetPSRFromPSTATE(AArch32_NonDebugState, 32) 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>} <Rd>, <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>} <Rd>, <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

<c> 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”:

R	M	M1	<banked_reg>
0	0	0000	R8_usr
0	0	0001	R9_usr
0	0	0010	R10_usr
0	0	0011	R11_usr
0	0	0100	R12_usr
0	0	0101	SP_usr
0	0	0110	LR_usr
0	0	0111	UNPREDICTABLE
0	0	1000	R8_fiq
0	0	1001	R9_fiq
0	0	1010	R10_fiq
0	0	1011	R11_fiq
0	0	1100	R12_fiq
0	0	1101	SP_fiq
0	0	1110	LR_fiq
0	0	1111	UNPREDICTABLE
0	1	0000	LR_irq
0	1	0001	SP_irq
0	1	0010	LR_svc
0	1	0011	SP_svc
0	1	0100	LR_abt
0	1	0101	SP_abt
0	1	0110	LR_und
0	1	0111	SP_und
0	1	10xx	UNPREDICTABLE
0	1	1100	LR_mon
0	1	1101	SP_mon
0	1	1110	ELR_hyp
0	1	1111	SP_hyp
1	0	0xxx	UNPREDICTABLE
1	0	10xx	UNPREDICTABLE
1	0	110x	UNPREDICTABLE
1	0	1110	SPSR_fiq
1	0	1111	UNPREDICTABLE
1	1	0000	SPSR_irq
1	1	0001	UNPREDICTABLE
1	1	0010	SPSR_svc
1	1	0011	UNPREDICTABLE
1	1	0100	SPSR_abt
1	1	0101	UNPREDICTABLE
1	1	0110	SPSR_und
1	1	0111	UNPREDICTABLE
1	1	10xx	UNPREDICTABLE
1	1	1100	SPSR_mon
1	1	1101	UNPREDICTABLE
1	1	1110	SPSR_hyp
1	1	1111	UNPREDICTABLE

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'  R[d] = SPSR_fiq<31:0>;
                when '10000'  R[d] = SPSR_irq<31:0>;
                when '10010'  R[d] = SPSR_svc<31:0>;
                when '10100'  R[d] = SPSR_abt<31:0>;
                when '10110'  R[d] = SPSR_und<31:0>;
                when '11100'
                    if !ELUsingAArch32(EL3) then AArch64.MonitorModeTrap();
                    R[d] = SPSR_mon;
                when '11110'  R[d] = SPSR_hyp<31:0>;
        else
            integer m;
            BankedRegisterAccessValid(SYSm, mode); // Check for UNPREDICTABLE cases
            case SYSm of
                when '00xxx'                       // Access the User mode registers
                    m = UInt(SYSm<2:0>) + 8;
                    R[d] = Rmode[m,M32_User];
                when '01xxx'                       // Access the FIQ mode registers
                    m = UInt(SYSm<2:0>) + 8;
                    R[d] = Rmode[m,M32_FIQ];
                when '1000x'                       // Access the IRQ mode registers
                    m = 14 - UInt(SYSm<0>);        // LR when SYSm<0> == 0, otherwise SP
                    R[d] = Rmode[m,M32_IRQ];
                when '1001x'                       // Access the Supervisor mode registers
                    m = 14 - UInt(SYSm<0>);        // LR when SYSm<0> == 0, otherwise SP
                    R[d] = Rmode[m,M32_Svc];
                when '1010x'                       // Access the Abort mode registers
                    m = 14 - UInt(SYSm<0>);        // LR when SYSm<0> == 0, otherwise SP
                    R[d] = Rmode[m,M32_Abort];
                when '1011x'                       // Access the Undefined mode registers
                    m = 14 - UInt(SYSm<0>);        // LR when SYSm<0> == 0, otherwise SP
                    R[d] = Rmode[m,M32_Undef];
                when '1110x'                       // Access Monitor registers
                    if !ELUsingAArch32(EL3) then AArch64.MonitorModeTrap();
                    m = 14 - UInt(SYSm<0>);        // LR when SYSm<0> == 0, otherwise SP
                    R[d] = Rmode[m,M32_Monitor];
                when '11110'                       // Access ELR_hyp register
                    R[d] = ELR_hyp;
                when '11111'                       // Access SP_hyp register
                    R[d] = Rmode[13,M32_Hyp];
```

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_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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

31	30	29	28	27	26	25	24	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	1	0	M1				(1)	(1)	(1)	(1)	(0)	(0)	1	M	0	0	0	0	Rn			
cond																															

A1

```
MSR{<c>}{<q>} <banked_reg>, <Rn>
```

```
n = UInt(Rn);  write_spsr = (R == '1');
if n == 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	0	0	R	Rn				1	0	(0)	0	M1				(0)	(0)	1	M	(0)	(0)	(0)	(0)

T1

```
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”:

R	M	M1	<banked_reg>
0	0	0000	R8_usr
0	0	0001	R9_usr
0	0	0010	R10_usr
0	0	0011	R11_usr
0	0	0100	R12_usr
0	0	0101	SP_usr
0	0	0110	LR_usr
0	0	0111	UNPREDICTABLE
0	0	1000	R8_fiq
0	0	1001	R9_fiq
0	0	1010	R10_fiq
0	0	1011	R11_fiq
0	0	1100	R12_fiq
0	0	1101	SP_fiq
0	0	1110	LR_fiq
0	0	1111	UNPREDICTABLE
0	1	0000	LR_irq
0	1	0001	SP_irq
0	1	0010	LR_svc
0	1	0011	SP_svc
0	1	0100	LR_abt
0	1	0101	SP_abt
0	1	0110	LR_und
0	1	0111	SP_und
0	1	10xx	UNPREDICTABLE
0	1	1100	LR_mon
0	1	1101	SP_mon
0	1	1110	ELR_hyp
0	1	1111	SP_hyp
1	0	0xxx	UNPREDICTABLE
1	0	10xx	UNPREDICTABLE
1	0	110x	UNPREDICTABLE
1	0	1110	SPSR_fiq
1	0	1111	UNPREDICTABLE
1	1	0000	SPSR_irq
1	1	0001	UNPREDICTABLE
1	1	0010	SPSR_svc
1	1	0011	UNPREDICTABLE
1	1	0100	SPSR_abt
1	1	0101	UNPREDICTABLE
1	1	0110	SPSR_und
1	1	0111	UNPREDICTABLE
1	1	10xx	UNPREDICTABLE
1	1	1100	SPSR_mon
1	1	1101	UNPREDICTABLE
1	1	1110	SPSR_hyp
1	1	1111	UNPREDICTABLE

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

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if PSTATE.EL == EL0 then
        UNPREDICTABLE;
    else
        mode = PSTATE.M;
        if write_spsr then
            SPSRaccessValid(SYSm, mode);             // Check for UNPREDICTABLE cases
            case SYSm of
                when '01110'  SPSR_fiq<31:0> = R[n];
                when '10000'  SPSR_irq<31:0> = R[n];
                when '10010'  SPSR_svc<31:0> = R[n];
                when '10100'  SPSR_abt<31:0> = R[n];
                when '10110'  SPSR_und<31:0> = R[n];
                when '11100'
                    if !ELUsingAArch32(EL3) then AArch64.MonitorModeTrap();
                    SPSR_mon<31:0> = R[n];
                when '11110'  SPSR_hyp<31:0> = R[n];
        else
            integer m;
            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'                       // Access Monitor registers
                    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 ELR_hyp register
                    ELR_hyp = R[n];
                when '11111'                       // Access SP_hyp register
                    Rmode[13,M32_Hyp] = R[n];
```

CONSTRAINED UNPREDICTABLE behavior

If `PSTATE.EL == EL0`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.

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

31 30 29 28	27	26	25	24	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	R	1	0	mask	(1)	(1)	(1)	(1)	imm12
cond														

A1 (!(R == 0 && mask == 0000))

```
MSR{<c>}{<q>} <spec_reg>, #<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:

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*.

<imm> Is an immediate value. See *Modified immediate constants in A32 instructions* for the range of values.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    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`.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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

31-28	27	26	25	24	23	22	21	20	19-16	15	14	13	12	11	10	9	8	7	6	5	4	3-0
!= 1111	0	0	0	1	0	R	1	0	mask	(1)	(1)	(1)	(1)	(0)	(0)	0	(0)	0	0	0	0	Rn
cond																						

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	3-0	15	14	13	12	11-8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	0	0	R	Rn	1	0	(0)	0	mask	(0)	(0)	0	(0)	(0)	(0)	(0)	(0)

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

```
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`.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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	S	Rd				(0)	(0)	(0)	(0)	Rm				1	0	0	1	Rn			
cond																															

Flag setting (S == 1)

```
MULS{<c>}{<q>} <Rd>, <Rn>{, <Rm>}
```

Not flag setting (S == 0)

```
MUL{<c>}{<q>} <Rd>, <Rn>{, <Rm>}
```

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '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
0	1	0	0	0	0	1	1	0	1	Rn			Rdm		

T1

```
MUL<c>{<q>} <Rdm>, <Rn>{, <Rdm>} // (Inside IT block)
```

```
MULS{<q>} <Rdm>, <Rn>{, <Rdm>} // (Outside IT block)
```

```
d = UInt(Rdm);  n = UInt(Rn);  m = UInt(Rdm);  setflags = !InITBlock();
```

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	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 = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  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*.

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

<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. If omitted, <Rd> is used.

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
    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_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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

31-28	27	26	25	24	23	22	21	20	19	18	17	16	15-12	11-0
!= 1111	0	0	1	1	1	1	1	S	(0)	(0)	(0)	(0)	Rd	imm12
cond														

MVN (S == 0)

```
MVN{<c>}{<q>} <Rd>, #<const>
```

MVNS (S == 1)

```
MVNS{<c>}{<q>} <Rd>, #<const>
```

```
d = UInt(Rd);  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	15	14-12	11-8	7-0
1	1	1	1	0	i	0	0	0	1	1	S	1	1	1	1	0	imm3	Rd	imm8

MVN (S == 0)

```
MVN{<c>}{<q>} <Rd>, #<const>
```

MVNS (S == 1)

```
MVNS{<c>}{<q>} <Rd>, #<const>
```

```
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 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.

<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 = 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
```

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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

31 30 29 28	27	26	25	24	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	1	S	(0)	(0)	(0)	(0)	Rd	imm5	stype	0	Rm
cond																	

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

```
MVN{<c>}{<q>} <Rd>, <Rm>, RRX
```

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

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

T1

```
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

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	1	1	1	1	(0)	imm3			Rd				imm2		stype		Rm			

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

```
MVN{<c>}{<q>} <Rd>, <Rm>, RRX
```

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

```
MVN<c>.W <Rd>, <Rm> // (Inside IT block, and <Rd>, <Rm> can be represented in T1)
```

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

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

```
MVNS{<c>}{<q>} <Rd>, <Rm>, RRX
```

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

```
MVNS.W <Rd>, <Rm> // (Outside IT block, and <Rd>, <Rm> can be represented in T1)
```

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

```
d = UInt(Rd);  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

<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”:

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.

Operation

```
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
```

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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

31 30 29 28	27	26	25	24	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	1	S	(0) (0) (0) (0)	Rd	Rs	0	stype	1	Rm
cond															

Flag setting (S == 1)

```
MVNS{<c>}{<q>} <Rd>, <Rm>, <shift> <Rs>
```

Not flag setting (S == 0)

```
MVN{<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;
```

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”:

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

<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 = 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.

Note

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
!= 1111				0	0	1	1	0	0	1	0	0	0	0	0	(1)	(1)	(1)	(1)	(0)	(0)	(0)	(0)	0	0	0	0	0	0	0	0
cond																															

A1

```
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	1	0	0	0	0	0	0	0	0

T1

```
NOP{<c>}{<q>}
```

```
// No additional decoding required
```

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	0	0	1	1	1	0	1	0	(1)	(1)	(1)	(1)	1	0	(0)	0	(0)	0	0	0	0	0	0	0	0	0	0	0

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

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	0	0	1	1	S	!= 1111	0	imm3	Rd	imm8
												Rn				

Flag setting (S == 1)

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

Not flag setting (S == 0)

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

```
if Rn == '1111' then SEE "MVN (immediate)";
d = UInt(Rd);  n = UInt(Rn);  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> 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			
												Rn																			

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>}
```

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

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

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

```
ORNS{<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

<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

<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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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	Rn	Rd	imm12
cond											

ORR (S == 0)

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

ORRS (S == 1)

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

```
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	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
												Rn				

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 = UInt(Rd);  n = UInt(Rn);  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. 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_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11–7	6–5	4	3–0
!= 1111	0	0	0	1	1	0	0	S	Rn	Rd	imm5	stype	0	Rm
cond														

ORR, rotate right with extend (S == 0 && imm5 == 00000 && 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 = 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	5–3	2–0
0	1	0	0	0	0	1	1	0	0	Rm	Rdn

T1

```
ORR<c>{<q>} {<Rdn>,} <Rdn>, <Rm> // (Inside IT block)

ORRS{<q>} {<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

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	0	S	!= 1111	(0)	imm3	Rd	imm2	stype	Rm
												Rn						

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”:

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

```
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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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

31 30 29 28	27 26 25 24 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	Rs	0	stype	1	Rm
cond										

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”:

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

<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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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

31 30 29 28	27 26 25 24 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 0	Rn	Rd	imm5	tb	0	1	Rm
cond								

PKHBT (tb == 0)

```
PKHBT{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, LSL #<imm>}
```

PKHTB (tb == 1)

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

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  tbform = (tb == '1');
(shift_t, shift_n) = DecodeImmShift(tb:'0', imm5);
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	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 0	0	Rn	(0)	imm3	Rd	imm2	tb	0	Rm
		S							T	

PKHBT (tb == 0)

```
PKHBT{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, LSL #<imm>} // (tbform == FALSE)
```

PKHTB (tb == 1)

```
PKHTB{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, ASR #<imm>} // (tbform == TRUE)
```

```
if S == '1' || T == '1' then UNDEFINED;
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  tbform = (tb == '1');
(shift_t, shift_n) = DecodeImmShift(tb:'0', 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.

<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:

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.

Note

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.

Note

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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

PLD (literal)

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) .

A1

31	30	29	28	27	26	25	24	23	22	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	(1)	0	1	1	1	1	1	(1)	(1)	(1)	(1)	imm12											

A1

```
PLD{<c>}{<q>} <label> // (Normal form)
```

```
PLD{<c>}{<q>} [PC, #{+/-}<imm>] // (Alternative form)
```

```
imm32 = ZeroExtend(imm12, 32);  add = (U == '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	1	1	0	0	0	U	0	(0)	1	1	1	1	1	1	1	1	1	imm12											

T1

```
PLD{<c>}{<q>} <label> // (Preferred syntax)
```

```
PLD{<c>}{<q>} [PC, #{+/-}<imm>] // (Alternative syntax)
```

```
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: 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.

+/- Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

U	+/-
0	-
1	+

<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

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = if add then (Align(PC,4) + imm32) else (Align(PC,4) - imm32);
    Hint_PreloadData(address);
```

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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

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	0	1	0	W	1	!= 1111	1	1	1	1	imm12
												Rn					

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

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	0	0	0	W	1	!= 1111	1	1	1	1	1	1	0	0	imm8
												Rn									

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(imm8, 32);  add = FALSE;  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*.

<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”:

U	+/-
0	-
1	+

\+ Specifies the offset is added to the base register.

<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] + imm32) else (R[n] - imm32);
    if is_pldw then
        Hint_PreloadDataForWrite(address);
    else
        Hint_PreloadData(address);
```

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

PLD, PLDW (register)

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	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

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	0	0	0	W	1	!= 1111				1	1	1	1	0	0	0	0	0	0	imm2		Rm			
												Rn																			

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”:

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.

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

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 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);
```

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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

31	30	29	28	27	26	25	24	23	22	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	U	1	0	1	Rn				(1)	(1)	(1)	(1)	imm12											

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

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	0	1	!= 1111				1	1	1	1	imm12											
												Rn																			

T1

```
PLI{<c>}{<q>} [<Rn> {, #{+}<imm>}]
```

```
if Rn == '1111' then SEE "encoding T3";
n = UInt(Rn);  imm32 = ZeroExtend(imm12, 32);  add = TRUE;
```

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	0	0	0	1	!= 1111				1	1	1	1	1	1	0	0	imm8							
												Rn																			

T2

```
PLI{<c>}{<q>} [<Rn> {, #-<imm>}]
```

```
if Rn == '1111' then SEE "encoding T3";
n = UInt(Rn);  imm32 = ZeroExtend(imm8, 32);  add = FALSE;
```

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	U	0	0	1	1	1	1	1	1	1	1	1	imm12											

T3

```
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*.

<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”:

U	+/-
0	-
1	+

\+ Specifies the offset is added to the base register.

<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 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);
```

PLI (register)

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	0	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]
```

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

```
PLI{<c>}{<q>} [<Rn>, {+/-}<Rm> {, <shift> #<amount>}]
```

```
n = UInt(Rn);  m = UInt(Rm);  add = (U == '1');
(shift_t, shift_n) = DecodeImmShift(stype, imm5);
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	0	1	0	0	0	1	!= 1111				1	1	1	1	0	0	0	0	0	0	imm2		Rm			
												Rn																			

T1

```
PLI{<c>}{<q>} [<Rn>, {+}<Rm> {, LSL #<amount>}]
```

```
if Rn == '1111' then SEE "PLI (immediate, literal)";
n = UInt(Rn);  m = UInt(Rm);  add = TRUE;
(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> — 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.

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

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 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);
    Hint_PreloadInstr(address);
```

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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

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

T1

```
POP{<c>}{<q>} <registers> // (Preferred syntax)
```

```
LDM{<c>}{<q>} SP!, <registers> // (Alternate syntax)
```

```
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.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[13];
    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 R[13] = R[13] + 4*BitCount(registers);
    if registers<13> == '1' then R[13] = bits(32) UNKNOWN;
```

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, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

It has encodings from the following instruction sets: A32 (A1) and T32 (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 0	0	1	0	1	1	1 1 0 1	register_list
cond					W		Rn	

A1

```
POP{<c>}{<q>} <registers>
```

is equivalent to

```
LDM{<c>}{<q>} SP!, <registers>
```

and is the preferred disassembly when `BitCount(register_list) > 1`.

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 0 1 0 0	0 1	0	1	1	1 1 0 1	P	M	register_list
			W		Rn			

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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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, any CONSTRAINED UNPREDICTABLE behavior, and any operational information 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	0	1	0	0	1	1 1 0 1	Rt	0 0 0 0 0 0 0 0 0 1 0 0
cond				P	U		W		Rn		imm12

Post-indexed

```
POP{<c>}{<q>} <single_register_list>
```

is equivalent to

```
LDR{<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	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	1	1 1 0 1	Rt	1	0	1	1	0 0 0 0 0 1 0 0
			Rn			P	U	W	imm8

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*.

<single_register_list> Is the general-purpose register <Rt> to be loaded 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. 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_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

PSSBB

Physical Speculative Store Bypass Barrier is a memory barrier that prevents speculative loads from bypassing earlier stores to the same physical address under certain conditions. For more information and details of the semantics, see *Physical Speculative Store Bypass Barrier (PSSBB)*.

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	0	1	1	1	(1)	(1)	(1)	(1)	(1)	(1)	(1)	(1)	(0)	(0)	(0)	(0)	0	1	0	0	0	1	0	0

A1

```
PSSBB{<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	0	(0)	0	(1)	(1)	(1)	(1)	0	1	0	0	0	1	0	0

T1

```
PSSBB{<q>}
```

```
if InITBlock() then UNPREDICTABLE;
```

Assembler Symbols

<q> See *Standard assembler syntax fields*.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    SpeculativeStoreBypassBarrierToPA();
```

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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

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

T1

```
PUSH{<c>}{<q>} <registers> // (Preferred syntax)
```

```
STMDB{<c>}{<q>} SP!, <registers> // (Alternate syntax)
```

```
registers = '0':M:'000000':register_list;  UnalignedAllowed = FALSE;
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 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 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.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[13] - 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();
    R[13] = R[13] - 4*BitCount(registers);
```

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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, any CONSTRAINED UNPREDICTABLE behavior, and any operational information 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
!= 1111	1	0	0	1	0	0	1	0	1	1	0	1	register_list
cond							W		Rn				

A1

```
PUSH{<c>}{<q>} <registers>
```

is equivalent to

```
STMDB{<c>}{<q>} SP!, <registers>
```

and is the preferred disassembly when `BitCount(register_list) > 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	0	1	0	0	1	0	0	1	0	1	1	0	1	(0)	M	register_list
										W		Rn				P		

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_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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, any CONSTRAINED UNPREDICTABLE behavior, and any operational information 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	0	1	0	1 1 0 1	Rt	0 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	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	Rt	1	1	0	1	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 *PSTATE*.Q to 1.

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

A1

31-28	27	26	25	24	23	22	21	20	19-16	15-12	11	10	9	8	7	6	5	4	3-0
!= 1111	0	0	0	1	0	0	0	0	Rn	Rd	(0)	(0)	(0)	(0)	0	1	0	1	Rm
cond																			

A1

```
QADD{<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	3-0	15	14	13	12	11-8	7	6	5	4	3-0
1	1	1	1	1	0	1	0	1	0	0	0	Rn	1	1	1	1	Rd	1	0	0	0	Rm

T1

```
QADD{<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();
    boolean sat;
    (R[d], sat) = SignedSatQ(SInt(R[m]) + SInt(R[n]), 32);
    if sat then
        PSTATE.Q = '1';
```

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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

31 30 29 28	27	26	25	24	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	0	Rn	Rd	(1)	(1)	(1)	(1)	0	0	0	1	Rm
cond																			

A1

```
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

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

T1

```
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

```
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);
```

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

QADD8

Saturating Add 8 performs four 8-bit integer additions, saturates the results to the 8-bit signed integer range $-2^7 <= x <= 2^7 - 1$, and writes the results to the destination register.

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

A1

31 30 29 28	27 26 25 24 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 0	Rn	Rd	(1)	(1)	(1)	(1)	1	0 0	1	Rm
cond												

A1

```
QADD8{<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	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	0	0	1	Rm

T1

```
QADD8{<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>   = 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_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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 $-2^{15} <= x <= 2^{15} - 1$, and writes the results to the destination register.

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

A1

31 30 29 28	27 26 25 24 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 0	Rn	Rd	(1)	(1)	(1)	(1)	0	0 1	1	Rm
cond												

A1

```
QASX{<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	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	0	1	Rm

T1

```
QASX{<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 = 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>  = SignedSat(diff, 16);
    R[d]<31:16> = SignedSat(sum, 16);
```

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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 $-2^{31} <= x <= 2^{31} - 1$. 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

31-28	27	26	25	24	23	22	21	20	19-16	15-12	11	10	9	8	7	6	5	4	3-0
!= 1111	0	0	0	1	0	1	0	0	Rn	Rd	(0)	(0)	(0)	(0)	0	1	0	1	Rm
cond																			

A1

```
QDADD{<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	3-0	15	14	13	12	11-8	7	6	5	4	3-0
1	1	1	1	1	0	1	0	1	0	0	0	Rn	1	1	1	1	Rd	1	0	0	1	Rm

T1

```
QDADD{<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);
    boolean sat2;
    (R[d], sat2)  = SignedSatQ(SInt(R[m]) + SInt(doubled), 32);
    if sat1 || sat2 then
        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																			

A1

```
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	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	1	0	1	1	Rm

T1

```
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);
    boolean sat2;
    (R[d], sat2)  = SignedSatQ(SInt(R[m]) - SInt(doubled), 32);
    if sat1 || sat2 then
        PSTATE.Q = '1';
```

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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 $-2^{15} <= x <= 2^{15} - 1$, and writes the results to the destination register.

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

A1

31 30 29 28	27 26 25 24 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 0	Rn	Rd	(1)	(1)	(1)	(1)	0	1	0	1	Rm
cond													

A1

```
QSAX{<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	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	1	Rm

T1

```
QSAX{<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();
    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>  = SignedSat(sum, 16);
    R[d]<31:16> = SignedSat(diff, 16);
```

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

QSUB

Saturating Subtract subtracts one register value from another register value, saturates the result to the 32-bit signed integer range $-2^{31} <= x <= 2^{31} - 1$, 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

31-28	27	26	25	24	23	22	21	20	19-16	15-12	11	10	9	8	7	6	5	4	3-0
!= 1111	0	0	0	1	0	0	1	0	Rn	Rd	(0)	(0)	(0)	(0)	0	1	0	1	Rm
cond																			

A1

```
QSUB{<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	3-0	15	14	13	12	11-8	7	6	5	4	3-0
1	1	1	1	1	0	1	0	1	0	0	0	Rn	1	1	1	1	Rd	1	0	1	0	Rm

T1

```
QSUB{<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();
    boolean sat;
    (R[d], sat) = SignedSatQ(SInt(R[m]) - SInt(R[n]), 32);
    if sat then
        PSTATE.Q = '1';
```

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

QSUB16

Saturating Subtract 16 performs two 16-bit integer subtractions, 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

31 30 29 28	27	26	25	24	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	0	Rn	Rd	(1)	(1)	(1)	(1)	0	1	1	1	Rm
cond																			

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

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

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
    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 $-2^{7} <= x <= 2^{7} - 1$, and writes the results to the destination register.
It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1

31 30 29 28	27 26 25 24 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 0	Rn	Rd	(1)	(1)	(1)	(1)	1	1	1	1	Rm
cond													

A1

```
QSUB8{<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	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 0 0	Rn	1 1 1 1	Rd	0	0	0	1	Rm

T1

```
QSUB8{<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>   = SignedSat(diff1, 8);
    R[d]<15:8>  = SignedSat(diff2, 8);
    R[d]<23:16> = SignedSat(diff3, 8);
    R[d]<31:24> = SignedSat(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

31-28	27	26	25	24	23	22	21	20	19	18	17	16	15-12	11	10	9	8	7	6	5	4	3-0
!= 1111	0	1	1	0	1	1	1	1	(1)	(1)	(1)	(1)	Rd	(1)	(1)	(1)	(1)	0	0	1	1	Rm
cond																						

A1

```
RBIT{<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	3-0	15	14	13	12	11-8	7	6	5	4	3-0
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 = 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*.

<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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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	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	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
```

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<31:24> = 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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

REV16

Byte-Reverse Packed Halfword reverses the byte order in each16-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>}{<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	1	Rm			Rd		

T1

```
REV16{<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	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>}{<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
```

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<31:24> = 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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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

31 30 29 28	27 26 25 24 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)	Rd	(1)	(1)	(1)	(1)	1	0 1 1	Rm
cond															

A1

```
REVSH{<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	1 1	Rm	Rd

T1

```
REVSH{<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	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 1	Rm

T2

```
REVSH{<c>}.W <Rd>, <Rm> // (<Rd>, <Rm> can be represented in T1)
```

```
REVSH{<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
```

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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

RFE, RFEDA, RFEDB, RFEIA, RFEIB

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

31	30	29	28	27	26	25	24	23	22	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	P	U	0	W	1	Rn				(0)	(0)	(0)	(0)	(1)	(0)	(1)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)

Decrement After (P == 0 && U == 0)

```
RFEDA{<c>}{<q>} <Rn>{!} // (Preferred syntax)
```

```
RFEFA{<c>}{<q>} <Rn>{!} // (Alternate syntax, Full Ascending stack)
```

Decrement Before (P == 1 && U == 0)

```
RFEDB{<c>}{<q>} <Rn>{!} // (Preferred syntax)
```

```
RFEEA{<c>}{<q>} <Rn>{!} // (Alternate syntax, Empty Ascending stack)
```

Increment After (P == 0 && U == 1)

```
RFE{IA}{<c>}{<q>} <Rn>{!} // (Preferred syntax)
```

```
RFEFD{<c>}{<q>} <Rn>{!} // (Alternate syntax, Full Descending stack)
```

Increment Before (P == 1 && U == 1)

```
RFEIB{<c>}{<q>} <Rn>{!} // (Preferred syntax)
```

```
RFEED{<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

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	0	W	1	Rn				(1)	(1)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)

T1

```
RFEDB{<c>}{<q>} <Rn>{!} // (Outside or last in IT block, preferred syntax)
```

```
RFEFA{<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

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	1	0	W	1	Rn				(1)	(1)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)

T2

```
RFE{IA}{<c>}{<q>} <Rn>{!} // (Outside or last in IT block, preferred syntax)
```

```
RFEFD{<c>}{<q>} <Rn>{!} // (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, any CONSTRAINED UNPREDICTABLE behavior, and any operational information 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) (0)	Rd	!= 00000	1 1	0	Rm
cond			S			imm5	stype		

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	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 1	(0)	imm3	Rd	imm2	1 1	Rm
		S						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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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, any CONSTRAINED UNPREDICTABLE behavior, and any operational information 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	0	1	1	0	1	0	(0)	(0)	(0)	(0)	Rd				Rs				0	1	1	1	Rm			
cond											S														stype						

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

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	1	Rs			Rdm		
						op									

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

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	0	0	1	1	0	Rm				1	1	1	1	Rd				0	0	0	0	Rs			
									stype		S																				

Not flag setting

```
ROR<c>.W {<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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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, any CONSTRAINED UNPREDICTABLE behavior, and any operational information 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	1	(0) (0) (0) (0)	Rd	!= 00000	1 1	0	Rm
cond			S			imm5	stype		

MOVS, shift or rotate by value

```
RORS{<c>}{<q>} {<Rd>,} <Rm>, #<imm>
```

is equivalent to

```
MOVS{<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	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 1	Rm
		S						stype	

MOVS, shift or rotate by value (!(imm3 == 000 && imm2 == 00))

```
RORS{<c>}{<q>} {<Rd>,} <Rm>, #<imm>
```

is equivalent to

```
MOVS{<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.

<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.

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, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

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

A1

31-28	27	26	25	24	23	22	21	20	19	18	17	16	15-12	11-8	7	6	5	4	3-0
!= 1111	0	0	0	1	1	0	1	1	(0)	(0)	(0)	(0)	Rd	Rs	0	1	1	1	Rm
cond								S								stype			

Flag setting

```
RORS{<c>}{<q>} {<Rd>,} <Rm>, <Rs>
```

is equivalent to

```
MOVS{<c>}{<q>} <Rd>, <Rm>, ROR <Rs>
```

and is always the preferred disassembly.

T1

15	14	13	12	11	10	9	8	7	6	5-3	2-0
0	1	0	0	0	0	0	1	1	1	Rs	Rdm
						op					

Rotate right

```
RORS{<q>} {<Rdm>,} <Rdm>, <Rs> // (Outside IT block)
```

is equivalent to

```
MOVS{<q>} <Rdm>, <Rdm>, ROR <Rs>
```

and is the preferred disassembly when `!InITBlock()`.

T2

15	14	13	12	11	10	9	8	7	6	5	4	3-0	15	14	13	12	11-8	7	6	5	4	3-0
1	1	1	1	1	0	1	0	0	1	1	1	Rm	1	1	1	1	Rd	0	0	0	0	Rs
									stype		S											

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

```
MOVS{<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, any CONSTRAINED UNPREDICTABLE behavior, and any operational information 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) (0)	Rd	0 0 0 0 0	1 1	0	Rm
cond			S			imm5	stype		

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	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 1	(0)	0 0 0	Rd	0 0	1 1	Rm
		S			imm3		imm2	stype	

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, any CONSTRAINED UNPREDICTABLE behavior, and any operational information 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	1	(0)	(0)	(0)	(0)	Rd	0	0	0	0	0	1	1	0	Rm
cond								S						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	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)	0	0	0	Rd	0	0	1	1	Rm
											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) gives the operational pseudocode for this instruction.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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 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

31-28	27-24	23-21	20	19-16	15-12	11-0
!= 1111	0 0 1 0	0 1 1	S	Rn	Rd	imm12
cond						

RSB (S == 0)

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

RSBS (S == 1)

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

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

T1

15-10	9-6	5-3	2-0
0 1 0 0 0 0	1 0 0 1	Rn	Rd

T1

```
RSB<c>{<q>} {<Rd>, }<Rn>, #0 // (Inside IT block)
```

```
RSBS{<q>} {<Rd>, }<Rn>, #0 // (Outside IT block)
```

```
d = UInt(Rd);  n = UInt(Rn);  setflags = !InITBlock();  imm32 = Zeros(32); // immediate = #0
```

T2

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

RSB (S == 0)

```
RSB<c>.W {<Rd>,} <Rn>, #0 // (Inside IT block)
```

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

RSBS (S == 1)

```
RSBS.W {<Rd>,} <Rn>, #0 // (Outside IT block)
```

```
RSBS{<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*.

<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 *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_<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.

<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(NOT(R[n]), 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;
```

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

31 30 29 28	27	26	25	24	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	S	Rn	Rd	imm5	stype	0	Rm
cond														

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

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

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

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

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

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

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

```
RSBS{<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	15	14 13 12	11 10 9 8	7 6	5 4	3 2 1 0
1	1	1	0	1	0	1	1	1	1	0	S	Rn	(0)	imm3	Rd	imm2	stype	Rm

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

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

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

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

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

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

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

```
RSBS{<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*.

<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 *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_<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 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.

<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”:

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 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

```
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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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

31 30 29 28	27 26 25 24	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	S	Rn	Rd	Rs	0	stype	1	Rm
cond										

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 = 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”:

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 = Shift(R[m], shift_t, shift_n, PSTATE.C);
    (result, nzcv) = AddWithCarry(NOT(R[n]), 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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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

31 30 29 28	27	26	25	24	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	1	S	Rn	Rd	imm12
cond											

RSC (S == 0)

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

RSCS (S == 1)

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

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

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 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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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”:

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

<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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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

31 30 29 28	27 26 25 24	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	Rs	0	stype	1	Rm
cond										

Flag setting (S == 1)

```
RSCS{<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

<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 = 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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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	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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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	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	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	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';
```

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_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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 *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	0 0 1	Rn	Rd	(1)	(1)	(1)	(1)	0	0 1	1	Rm
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

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 0 1	0 1 0	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 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.

SB

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
(FEAT_SB)

31	30	29	28	27	26	25	24	23	22	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	0	1	1	1	(1)	(1)	(1)	(1)	(1)	(1)	(1)	(1)	(0)	(0)	(0)	(0)	0	1	1	1	(0)	(0)	(0)	(0)

A1

```
SB{<q>}
```

```
// No additional decoding required
```

T1
(FEAT_SB)

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	0	(0)	0	(1)	(1)	(1)	(1)	0	1	1	1	(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();
```

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

31-28	27	26	25	24	23	22	21	20	19-16	15-12	11-0
!= 1111	0	0	1	0	1	1	0	S	Rn	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	3-0	15	14-12	11-8	7-0
1	1	1	1	0	i	0	1	0	1	1	S	Rn	0	imm3	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*.

<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 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, 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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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

31-28	27	26	25	24	23	22	21	20	19-16	15-12	11-7	6-5	4	3-0
!= 1111	0	0	0	0	1	1	0	S	Rn	Rd	imm5	stype	0	Rm
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 = 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	5-3	2-0
0	1	0	0	0	0	0	1	1	0	Rm	Rdn

T1

```
SBC<c>{<q>} {<Rdn>,} <Rdn>, <Rm> // (Inside IT block)
```

```
SBCS{<q>} {<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

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	1	0	1	1	S	Rn				(0)	imm3			Rd				imm2		stype		Rm			

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>.W {<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”:

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.

Operation

```
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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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

31 30 29 28	27 26 25 24	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	Rn	Rd	Rs	0	stype	1	Rm
cond										

Flag setting (S == 1)

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

Not flag setting (S == 0)

```
SBC{<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”:

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 = 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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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

```
SBFX{<c>}{<q>} <Rd>, <Rn>, #<lsb>, #<width>
```

```
d = UInt(Rd);  n = UInt(Rn);
lsbit = UInt(lsb);  widthminus1 = UInt(widthm1);
msbit = lsbit + widthminus1;
if d == 15 || n == 15 then UNPREDICTABLE;
if msbit > 31 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `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.

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 1	0	0	Rn	0	imm3	Rd	imm2	(0)	widthm1

T1

```
SBFX{<c>}{<q>} <Rd>, <Rn>, #<lsb>, #<width>
```

```
d = UInt(Rd);  n = UInt(Rn);
lsbit = UInt(imm3:imm2);  widthminus1 = UInt(widthm1);
msbit = lsbit + widthminus1;
if d == 15 || n == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
if msbit > 31 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `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.

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

```
if ConditionPassed() then
    EncodingSpecificOperations();
    R[d] = SignExtend(R[n]<msbit:lsbit>, 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_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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

31 30 29 28	27 26 25 24 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 1	Rd	(1) (1) (1) (1)	Rm	0 0 0	1	Rn
cond				Ra				

A1

```
SDIV{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  a = UInt(Ra);
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 executes as described, and the register specified by Ra becomes 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 1	0 0 1	Rn	(1) (1) (1) (1)	Rd	1 1 1 1	Rm
			Ra			

T1

```
SDIV{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  a = UInt(Ra);
// Armv8-A removes UNPREDICTABLE for R13
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 executes as described, 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*.
<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

```
if ConditionPassed() then
    EncodingSpecificOperations();
    integer result;
    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_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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

31 30 29 28	27	26	25	24	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	0	Rn	Rd	(1)	(1)	(1)	(1)	1	0	1	1	Rm
cond																			

A1

```
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	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	1	0	0	0	Rm

T1

```
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();
    R[d]<7:0>   = if PSTATE.GE<0> == '1' then R[n]<7:0>   else R[m]<7:0>;
    R[d]<15:8>  = if PSTATE.GE<1> == '1' then R[n]<15:8>  else R[m]<15:8>;
    R[d]<23:16> = if PSTATE.GE<2> == '1' then R[n]<23:16> else R[m]<23:16>;
    R[d]<31:24> = if PSTATE.GE<3> == '1' then R[n]<31:24> else R[m]<31:24>;
```

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

SETEND

Set Endianness writes a new value to *PSTATE*.E.
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	0	0	0	0	(0)	(0)	(0)	1	(0)	(0)	(0)	(0)	(0)	(0)	E	(0)	0	0	0	0	(0)	(0)	(0)	(0)

A1

```
SETEND{<q>} <endian_specifier> // (Cannot be conditional)
```

```
set_bigend = (E == '1');
```

T1

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	0	(1)	E	(0)	(0)	(0)

T1

```
SETEND{<q>} <endian_specifier> // (Not permitted in IT block)
```

```
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”:

E	<endian_specifier>
0	LE
1	BE

Operation

```
EncodingSpecificOperations();
AArch32.CheckSETENDEnabled();
PSTATE.E = if set_bigend then '1' else '0';
```

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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
(FEAT_PAN)

31	30	29	28	27	26	25	24	23	22	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	0	0	0	1	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	imm1	(0)	0	0	0	0	(0)	(0)	(0)	(0)

A1

```
SETPAN{<q>} #<imm> // (Cannot be conditional)
```

```
if !HavePANExt() then UNDEFINED;
value = imm1;
```

T1
(FEAT_PAN)

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	0	0	(1)	imm1	(0)	(0)	(0)

T1

```
SETPAN{<q>} #<imm> // (Not permitted in IT block)
```

```
if InITBlock() then UNPREDICTABLE;
if !HavePANExt() then UNDEFINED;
value = imm1;
```

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_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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

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

A1

```
SEV{<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	1	0	1	0	0	0	0	0	0

T1

```
SEV{<c>}{<q>}
```

```
// No additional decoding required
```

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	0	0	1	1	1	0	1	0	(1)	(1)	(1)	(1)	1	0	(0)	0	(0)	0	0	0	0	0	0	0	0	1	0	0

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

```
if ConditionPassed() then
    EncodingSpecificOperations();
    SendEvent();
```

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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

31-28	27	26	25	24	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	1	0	0	0	0	0	(1)	(1)	(1)	(1)	(0)	(0)	(0)	(0)	0	0	0	0	0	1	0	1
cond																												

A1

```
SEVL{<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	1	0	1	0	1	0	0	0	0

T1

```
SEVL{<c>}{<q>}
```

```
// No additional decoding required
```

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	0	0	1	1	1	0	1	0	(1)	(1)	(1)	(1)	1	0	(0)	0	(0)	0	0	0	0	0	0	0	0	1	0	1

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

```
if ConditionPassed() then
    EncodingSpecificOperations();
    SendEventLocal();
```

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

SHADD16

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
cond																			

A1

```
SHADD16{<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	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	1	0	Rm

T1

```
SHADD16{<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<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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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

31 30 29 28	27	26	25	24	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)	1	0	0	1	Rm
cond																			

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

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

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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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

31 30 29 28	27 26 25 24 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	1	1	Rm
cond													

A1

```
SHASX{<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	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

T1

```
SHASX{<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 = 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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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

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

A1

```
SHSAX{<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	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	1	0	Rm			

T1

```
SHSAX{<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();
    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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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

```
SHSUB16{<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	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	0	1	Rn	1	1	1	1	Rd	0	0	1	0	Rm

T1

```
SHSUB16{<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<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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

SHSUB8

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

31 30 29 28	27 26 25 24 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)	1	1 1	1	Rm
cond												

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

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

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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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	1	1	1	imm4			
cond																															

A1

```
SMC{<c>}{<q>} {#}<imm4>
```

```
// imm4 is for assembly/disassembly only and is ignored by hardware
```

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	1	1	1	1	1	1	1	imm4				1	0	0	0	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)

T1

```
SMC{<c>}{<q>} {#}<imm4>
```

```
// imm4 is for assembly/disassembly only and is ignored by hardware
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.

Operation

```
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 CurrentSecurityState() == SS_Secure 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;

    if !ELUsingAArch32(EL3) then
        AArch64.CallSecureMonitor(Zeros(16));
    else
        AArch32.TakeSMCException();
```

CONSTRAINED UNPREDICTABLE behavior

If `SCR.SCD == '1' && CurrentSecurityState() == SS_Secure`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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	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 = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  a = UInt(Ra);
n_high = (N == '1');  m_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	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	1	Rn	!= 1111	Rd	0	0	N	M	Rm
													Ra						

SMLABB (N == 0 && M == 0)

```
SMLABB{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>
```

SMLABT (N == 0 && M == 1)

```
SMLABT{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>
```

SMLATB (N == 1 && M == 0)

```
SMLATB{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>
```

SMLATT (N == 1 && M == 1)

```
SMLATT{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>
```

```
if Ra == '1111' then SEE "SMULBB, SMULBT, SMULTB, SMULTT";
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  a = UInt(Ra);
n_high = (N == '1');  m_high = (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 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
    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) + SInt(R[a]);
    R[d] = result<31:0>;
    if result != SInt(result<31:0>) then  // Signed overflow
        PSTATE.Q = '1';
```

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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>
```

SMLADX (M == 1)

```
SMLADX{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>
```

```
if Ra == '1111' then SEE "SMUAD";
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	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>
```

SMLADX (M == 1)

```
SMLADX{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>
```

```
if Ra == '1111' then SEE "SMUAD";
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.

<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

31 30 29 28	27 26 25 24	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	RdHi	RdLo	Rm	1 0 0 1	Rn
cond								

Flag setting (S == 1)

```
SMLALS{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>
```

Not flag setting (S == 0)

```
SMLAL{<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;
```

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	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	0 0 0 0	Rm

T1

```
SMLAL{<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;
```

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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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 = UInt(RdLo);  dHi = UInt(RdHi);  n = UInt(Rn);  m = UInt(Rm);
n_high = (N == '1');  m_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	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>
```

```
dLo = UInt(RdLo);  dHi = UInt(RdHi);  n = UInt(Rn);  m = UInt(Rm);
n_high = (N == '1');  m_high = (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>	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 <x>), 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

```
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) + 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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

SMLALD, SMLALDX

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	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

```
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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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 *PSTATE*.Q to 1. No overflow can 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	1	0	Rd	Ra	Rm	1	M	0	0	Rn
cond																

SMLAWB (M == 0)

```
SMLAWB{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>
```

SMLAWT (M == 1)

```
SMLAWT{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>
```

```
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;
```

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	1	1	Rn	!= 1111	Rd	0	0	0	M	Rm
													Ra						

SMLAWB (M == 0)

```
SMLAWB{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>
```

SMLAWT (M == 1)

```
SMLAWT{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>
```

```
if Ra == '1111' then SEE "SMULWB, SMULWT";
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
```

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.

<Ra> Is the third general-purpose source register holding the addend, encoded in the "Ra" field.

Operation

```
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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

SMLSD, SMLSDX

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	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	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	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.
<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';
```

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_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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	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 UPREDICTABLE 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_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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

31 30 29 28	27	26	25	24	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	1	Rd	!= 1111	Rm	0	0	R	1	Rn
cond										Ra						

SMMLA (R == 0)

```
SMMLA{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>
```

SMMLAR (R == 1)

```
SMMLAR{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>
```

```
if Ra == '1111' then SEE "SMMUL";
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  a = UInt(Ra);  round = (R == '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	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	0	1	Rn	!= 1111	Rd	0	0	0	R	Rm
													Ra						

SMMLA (R == 0)

```
SMMLA{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>
```

SMMLAR (R == 1)

```
SMMLAR{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>
```

```
if Ra == '1111' then SEE "SMMUL";
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  a = UInt(Ra);  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 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_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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

31-28	27	26	25	24	23	22	21	20	19-16	15-12	11-8	7	6	5	4	3-0
!= 1111	0	1	1	1	0	1	0	1	Rd	Ra	Rm	1	1	R	1	Rn
cond																

SMMLS (R == 0)

```
SMMLS{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>
```

SMMLSR (R == 1)

```
SMMLSR{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>
```

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  a = UInt(Ra);  round = (R == '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	3-0	15-12	11-8	7	6	5	4	3-0
1	1	1	1	1	0	1	1	0	1	1	0	Rn	Ra	Rd	0	0	0	R	Rm

SMMLS (R == 0)

```
SMMLS{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>
```

SMMLSR (R == 1)

```
SMMLSR{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>
```

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  a = UInt(Ra);  round = (R == '1');
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 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_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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

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

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

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	1	0	1	Rn				1	1	1	1	Rd				0	0	0	R	Rm			

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 ConditionPassed() then
    EncodingSpecificOperations();
    result = 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_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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) .

A1

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

SMUAD (M == 0)

```
SMUAD{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

SMUADX (M == 1)

```
SMUADX{<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

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	1	0	Rn				1	1	1	1	Rd				0	0	0	M	Rm			

SMUAD (M == 0)

```
SMUAD{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

SMUADX (M == 1)

```
SMUADX{<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

<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;
    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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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

31 30 29 28	27	26	25	24	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	Rd	(0)	(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 = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
n_high = (N == '1');  m_high = (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	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	1	Rn	1	1	1	1	Rd	0	0	N	M	Rm

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 = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
n_high = (N == '1');  m_high = (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 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

```
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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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			
cond																															

Flag setting (S == 1)

```
SMULLS{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>
```

Not flag setting (S == 0)

```
SMULL{<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;
```

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	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	Rn				RdLo				RdHi				0	0	0	0	Rm			

T1

```
SMULL{<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;
```

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 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 = 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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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

31 30 29 28	27 26 25 24 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	Rd	(0) (0) (0) (0)	Rm	1	M	1	0	Rn
cond											

SMULWB (M == 0)

```
SMULWB{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

SMULWT (M == 1)

```
SMULWT{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  m_high = (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	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 1	Rn	1 1 1 1	Rd	0 0	0	M	Rm

SMULWB (M == 0)

```
SMULWB{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

SMULWT (M == 1)

```
SMULWT{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  m_high = (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 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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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

31	30	29	28	27	26	25	24	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				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

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	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

<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;
    R[d] = result<31:0>;
    // Signed overflow cannot occur
```

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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.

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	0	0	P	U	1	W	0	(1)	(1)	(0)	(1)	(0)	(0)	(0)	(0)	(0)	(1)	(0)	(1)	(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	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	W	0	(1)	(1)	(0)	(1)	(1)	(1)	(0)	(0)	(0)	(0)	(0)	(0)	(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	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)	(1)	(0)	(1)	(1)	(1)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	mode				

T2

```
SRS{IA}{<c>}{<q>} SP{!}, #<mode>
```

```
wback = (W == '1');  increment = TRUE;  wordhigher = 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

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*.
!	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.
<mode>	Is the number of the mode whose Banked SP is used as the base register, encoded in the "mode" field. For details of PE modes and their numbers see *AArch32 PE mode descriptions*.

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.

Operation

```
if CurrentInstrSet() == InstrSet_A32 then
    if ConditionPassed() then
        EncodingSpecificOperations();
        if PSTATE.EL == EL2 then          // UNDEFINED at EL2
            UNDEFINED;

        // Check for UNPREDICTABLE cases. The definition of UNPREDICTABLE does not permit these
        // to be security holes
        if PSTATE.M IN {M32_User,M32_System} then
            UNPREDICTABLE;
        elsif mode == M32_Hyp then        // Check for attempt to access Hyp mode SP
            UNPREDICTABLE;
        elsif mode == M32_Monitor then    // Check for attempt to access Monitor mode SP
            if !HaveEL(EL3) || CurrentSecurityState() != SS_Secure then
                UNPREDICTABLE;
            elsif !ELUsingAArch32(EL3) then
                AArch64.MonitorModeTrap();
        elsif BadMode(mode) then
            UNPREDICTABLE;

        base = Rmode[13,mode];
        address = if increment then base else base-8;
        if wordhigher then address = address+4;
        MemA[address,4]   = LR;
        MemA[address+4,4] = SPSR[];
        if wback then Rmode[13,mode] = if increment then base+8 else base-8;
else
    if ConditionPassed() then
        EncodingSpecificOperations();
        if PSTATE.EL == EL2 then          // UNDEFINED at EL2
            UNDEFINED;

        // Check for UNPREDICTABLE cases. The definition of UNPREDICTABLE does not permit these
        // to be security holes
        if PSTATE.M IN {M32_User,M32_System} then
            UNPREDICTABLE;
        elsif mode == M32_Hyp then        // Check for attempt to access Hyp mode SP
            UNPREDICTABLE;
        elsif mode == M32_Monitor then    // Check for attempt to access Monitor mode SP
            if !HaveEL(EL3) || CurrentSecurityState() != SS_Secure then
                UNPREDICTABLE;
            elsif !ELUsingAArch32(EL3) then
                AArch64.MonitorModeTrap();
        elsif BadMode(mode) then
            UNPREDICTABLE;

        base = Rmode[13,mode];
        address = if increment then base else base-8;
        if wordhigher then address = address+4;
        MemA[address,4]   = LR;
        MemA[address+4,4] = SPSR[];
        if wback then Rmode[13,mode] = if increment then base+8 else base-8;
```

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.

If `mode == M32_Hyp`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.

If `mode == M32_Monitor && (!HaveEL(EL3) || CurrentSecurityState() != SS_Secure)`, 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

31-28	27	26	25	24	23	22	21	20-16	15-12	11-7	6	5	4	3-0
!= 1111	0	1	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	3-0	15	14-12	11-8	7-6	5	4-0
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) == '00000' then SEE "SSAT16";
d = UInt(Rd);  n = UInt(Rn);  saturate_to = UInt(sat_imm)+1;
(shift_t, shift_n) = DecodeImmShift(sh:'0', imm3: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

```
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';
```

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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	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)	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_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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

31	30	29	28	27	26	25	24	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>
```

```
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	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>
```

```
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();
    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 that prevents speculative loads from bypassing earlier stores to the same virtual address under certain conditions. For more information and details of the semantics, see *Speculative Store Bypass Barrier (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	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	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	(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

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

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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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

31 30 29 28	27	26	25	24	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)	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

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	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

<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<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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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

31 30 29 28	27	26	25	24	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	0	W	0	Rn	0	1	0	1	1	1	1	0	imm8
cond																		

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;
```

CONSTRAINED 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

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	P	U	0	W	0	Rn	0	1	0	1	1	1	1	0	imm8

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;
```

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 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

<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”:

U	+/-
0	-
1	+

<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 read from DBGDTRRXint.
    AArch32.SysRegRead(cp, ThisInstr(), address<31:0>);
    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_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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																			

A1

```
STL{<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	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	1	0	(1) (1) (1) (1)

T1

```
STL{<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, 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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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)	(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	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

```
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];
    MemO[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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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

31 30 29 28	27	26	25	24	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	Rd	(1)	(1)	1	0	1	0	0	1	Rt
cond																			

A1

```
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 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)	1	1	1	0	Rd

T1

```
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 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 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,4) then
        MemO[address, 4] = R[t];
        R[d] = ZeroExtend('0', 32);
    else
        R[d] = ZeroExtend('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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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	0	1	1	1	0	0	Rn	Rd	(1)	(1)	1	0	1	0	0	1	Rt
cond																			

A1

```
STLEXB{<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)	1	1	0	0	Rd

T1

```
STLEXB{<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.

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
        MemO[address, 1] = R[t]<7:0>;
        R[d] = ZeroExtend('0', 32);
    else
        R[d] = ZeroExtend('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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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 = 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;
```

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	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

T1

```
STLEXD{<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;
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 = R[n];
    // Create doubleword to store such that R[t] will be stored at address and R[t2] at address+4.
    value = if BigEndian(AccessType_GPR) then R[t]:R[t2] else R[t2]:R[t];
    if AArch32.ExclusiveMonitorsPass(address, 8) then
        MemO[address, 8] = value;
        R[d] = ZeroExtend('0', 32);
    else
        R[d] = ZeroExtend('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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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	1	0	Rn	Rd	(1)	(1)	1	0	1	0	0	1	Rt
cond																			

A1

```
STLEXH{<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)	1	1	0	1	Rd

T1

```
STLEXH{<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.

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', 32);
    else
        R[d] = ZeroExtend('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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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

31-28	27	26	25	24	23	22	21	20	19-16	15	14	13	12	11	10	9	8	7	6	5	4	3-0
!= 1111	0	0	0	1	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	3-0	15-12	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	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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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
!= 1111	1	0	0	P	U	1	(0)	0	Rn	register_list
cond										

A1

```
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.

<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, 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
                MemS[address,4] = Rmode[i, M32_User];
                address = address + 4;
        if registers<15> == '1' then
            MemS[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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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

31–28	27	26	25	24	23	22	21	20	19–16	15–0
!= 1111	1	0	0	0	1	0	W	0	Rn	register_list
cond										

A1

```
STM{IA}{<c>}{<q>} <Rn>{!}, <registers> // (Preferred syntax)
```

```
STMEA{<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;
```

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	10–8	7–0
1	1	0	0	0	Rn	register_list

T1

```
STM{IA}{<c>}{<q>} <Rn>!, <registers> // (Preferred syntax)
```

```
STMEA{<c>}{<q>} <Rn>!, <registers> // (Alternate syntax, Empty Ascending stack)
```

```
n = UInt(Rn);  registers = '00000000':register_list;  wback = TRUE;
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 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*.

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	0	1	0	0	0	1	0	W	0	Rn				(0)	M	register_list													
																P															

T2

```
STM{IA}{<c>}.W <Rn>{!}, <registers> // (Preferred syntax, if <Rn>, '!' and <registers> can be represented

STMEA{<c>}.W <Rn>{!}, <registers> // (Alternate syntax, Empty Ascending stack, if <Rn>, '!' and <registers

STM{IA}{<c>}{<q>} <Rn>{!}, <registers> // (Preferred syntax)

STMEA{<c>}{<q>} <Rn>{!}, <registers> // (Alternate syntax, Empty Ascending stack)
```

```
n = UInt(Rn);  registers = P:M:register_list;  wback = (W == '1');
if n == 15 || BitCount(registers) < 2 then UNPREDICTABLE;
if wback && registers<n> == '1' then UNPREDICTABLE;
if registers<13> == '1' then UNPREDICTABLE;
if registers<15> == '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 the instruction specifies writeback, the modification to the base address on writeback might differ from the number of registers stored.

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 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 wback && 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 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 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 == 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*.

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

```
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
                MemS[address,4] = bits(32) UNKNOWN;  // Only possible for encodings T1 and A1
            else
                MemS[address,4] = R[i];
            address = address + 4;
    if registers<15> == '1' then  // Only possible for encoding A1
        MemS[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.

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

31 30 29 28	27 26 25	24	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	0	W	0	Rn	register_list
cond								

A1

```
STMDA{<c>}{<q>} <Rn>{!}, <registers> // (Preferred syntax)
```

```
STMED{<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;
```

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 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 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.

! — 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 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*BitCount(registers) + 4;
    for i = 0 to 14
        if registers<i> == '1' then
            if i == n && wback && i != LowestSetBit(registers) then
                MemS[address,4] = bits(32) UNKNOWN;
            else
                MemS[address,4] = R[i];
            address = address + 4;
    if registers<15> == '1' then
        MemS[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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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
cond								

A1

```
STMDB{<c>}{<q>} <Rn>{!}, <registers> // (Preferred syntax)

STMFD{<c>}{<q>} <Rn>{!}, <registers> // (Alternate syntax, Full Descending stack)
```

```
n = UInt(Rn);  registers = register_list;  wback = (W == '1');
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. If the instruction specifies writeback, the modification to the base address on writeback might differ from the number of registers stored.

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	1	0	0	W	0	Rn	(0)	M	register_list
							P		

T1

```
STMDB{<c>}{<q>} <Rn>{!}, <registers> // (Preferred syntax)

STMFD{<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 then UNPREDICTABLE;
if wback && registers<n> == '1' then UNPREDICTABLE;
if registers<13> == '1' then UNPREDICTABLE;
if registers<15> == '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 the instruction specifies writeback, the modification to the base address on writeback might differ from the number of registers stored.

If `wback && 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 `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 instruction operates as an STM 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 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 `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

<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-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

Alias	Of variant	Is preferred when
PUSH (multiple registers)	T1	`W == '1' && Rn == '1101' && BitCount(M:register_list) > 1`
PUSH (multiple registers)	A1	`W == '1' && Rn == '1101' && BitCount(register_list) > 1`

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n] - 4*BitCount(registers);
    for i = 0 to 14
        if registers<i> == '1' then
            if i == n && wback && i != LowestSetBit(registers) then
                MemS[address,4] = bits(32) UNKNOWN;  // Only possible for encoding A1
            else
                MemS[address,4] = R[i];
            address = address + 4;
    if registers<15> == '1' then  // Only possible for encoding A1
        MemS[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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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	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 = UInt(Rn);  registers = register_list;  wback = (W == '1');
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. 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 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.
!	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 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
                MemS[address,4] = bits(32) UNKNOWN;
            else
                MemS[address,4] = R[i];
            address = address + 4;
    if registers<15> == '1' then
        MemS[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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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

31 30 29 28	27	26	25	24	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	0	Rn	Rt	imm12
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

15 14 13	12	11	10 9 8 7 6	5 4 3	2 1 0
0 1 1	0	0	imm5	Rn	Rt

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

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

T2

```
STR{<c>}{<q>} <Rt>, [SP{, #{+}<imm>}]
```

```
t = UInt(Rt);  n = 13;  imm32 = ZeroExtend(imm8:'00', 32);
index = TRUE;  add = TRUE;  wback = FALSE;
```

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	0	1	1	0	0	!= 1111				Rt				imm12											
												Rn																			

T3

```
STR{<c>}.W <Rt>, [<Rn> {, #{+}<imm>}] // (<Rt>, <Rn>, <imm> can be represented in T1 or T2)
```

```
STR{<c>}{<q>} <Rt>, [<Rn> {, #{+}<imm>}]
```

```
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;
```

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

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	0	0	1	0	0	!= 1111				Rt				1	P	U	W	imm8							
												Rn																			

Offset (P == 1 && U == 0 && W == 0)

```
STR{<c>}{<q>} <Rt>, [<Rn> {, #-<imm>}]
```

Post-indexed (P == 0 && W == 1)

```
STR{<c>}{<q>} <Rt>, [<Rn>], #{+/-}<imm>
```

Pre-indexed (P == 1 && W == 1)

```
STR{<c>}{<q>} <Rt>, [<Rn>, #{+/-}<imm>]!
```

```
if P == '1' && U == '1' && W == '0' then SEE "STRT";
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;
```

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 `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”:

U	+/-
0	-
1	+

\+ 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

Alias	Of variant	Is preferred when
PUSH (single register)	A1 (pre-indexed)	`P == '1' && U == '0' && W == '1' && Rn == '1101' && imm12 == '000000000100'`
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 (R[n] + imm32) else (R[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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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

31-28	27	26	25	24	23	22	21	20	19-16	15-12	11-7	6-5	4	3-0
!= 1111	0	1	1	P	U	0	W	0	Rn	Rt	imm5	stype	0	Rm
cond														

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";
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 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	8-6	5-3	2-0
0	1	0	1	0	0	0	Rm	Rn	Rt

T1

```
STR{<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);
```

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	0	0	1	0	0	!= 1111				Rt				0	0	0	0	0	0	imm2		Rm			
												Rn																			

T2

```
STR{<c>}.W <Rt>, [<Rn>, {+}<Rm>] // (<Rt>, <Rn>, <Rm> can be represented in T1)
```

```
STR{<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) = (SRType_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> — 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”:

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.

<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];
    bits(32) data;
    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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11–0
!= 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";
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm12, 32);
index = (P == '1');  add = (U == '1');  wback = (P == '0') || (W == '1');
if t == 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	10–6	5–3	2–0
0	1	1	1	0	imm5	Rn	Rt

T1

```
STRB{<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	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	0	0	0	!= 1111	Rt	imm12
												Rn		

T2

```
STRB{<c>}.W <Rt>, [<Rn> {, #{+}<imm>}] // (<Rt>, <Rn>, <imm> can be represented in T1)
```

```
STRB{<c>}{<q>} <Rt>, [<Rn> {, #{+}<imm>}]
```

```
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
```

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	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	!= 1111	Rt	1	P	U	W	imm8
												Rn						

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;
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
```

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.

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”:

U	+/-
0	-
1	+

+	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.

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 = 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;
```

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) = (SRType_LSL, 0);
```

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	0	0	0	0	0	!= 1111				Rt				0	0	0	0	0	0	imm2		Rm			
												Rn																			

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) = (SRType_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”:

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.

<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];
    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_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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

31 30 29 28	27	26	25	24	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	0	Rn	Rt	imm12
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

31 30 29 28	27	26	25	24	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	0	Rn	Rt	imm5	stype	0	Rm
cond														

A2

```
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

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	0	0	0	0	0	!= 1111				Rt				1	1	1	0	imm8							
												Rn																			

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*.

<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”:

U	+/-
0	-
1	+

For encoding A2: specifies the index register is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

U	+/-
0	-
1	+

<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
    EncodingSpecificOperations();
    if PSTATE.EL == EL2 then UNPREDICTABLE;                // Hyp mode
    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;
    MemU_unpriv[address,1] = R[t]<7:0>;
    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 `STRB` (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_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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
cond																

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;
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 == 15 || n == t || n == t2) then UNPREDICTABLE;
if t2 == 15 then UNPREDICTABLE;
```

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.

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'.

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

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	P	U	1	W	0	!= 1111	Rt	Rt2	imm8
												Rn			

Offset (P == 1 && W == 0)

```
STRD{<c>}{<q>} <Rt>, <Rt2>, [<Rn> {, #{+/-}<imm>}]
```

Post-indexed (P == 0 && W == 1)

```
STRD{<c>}{<q>} <Rt>, <Rt2>, [<Rn>], #{+/-}<imm>
```

Pre-indexed (P == 1 && W == 1)

```
STRD{<c>}{<q>} <Rt>, <Rt2>, [<Rn>, #{+/-}<imm>]!
```

```
if P == '0' && W == '0' then SEE "Related encodings";
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 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”:

U	+/-
0	-
1	+

<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

```
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 IsAligned(address, 8) then
        bits(64) data;
        if BigEndian(AccessType_GPR) then
            data<63:32> = R[t];
            data<31:0> = R[t2];
        else
            data<31:0> = R[t];
            data<63:32> = R[t2];
        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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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

31 30 29 28	27	26	25	24	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	1	1	1	Rm
cond																			

Offset (P == 1 && W == 0)

```
STRD{<c>}{<q>} <Rt>, <Rt2>, [<Rn>, {+/-}<Rm>]
```

Post-indexed (P == 0 && W == 0)

```
STRD{<c>}{<q>} <Rt>, <Rt2>, [<Rn>], {+/-}<Rm>
```

Pre-indexed (P == 1 && W == 1)

```
STRD{<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 then UNPREDICTABLE;
if wback && (n == 15 || n == t || n == t2) then UNPREDICTABLE;
```

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.

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'.

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 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

<c> See *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<Rt> Is the first general-purpose register to be transferred, encoded in the "Rt" field. This register must be even-numbered and not R14.

<Rt2> Is the second general-purpose register to be transferred. This register must be <R(t+1)>.

<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”:

U	+/-
0	-
1	+

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

Operation

```
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 IsAligned(address, 8) then
        bits(64) data;
        if BigEndian(AccessType_GPR) then
            data<63:32> = R[t];
            data<31:0> = R[t2];
        else
            data<31:0> = R[t];
            data<63:32> = R[t2];
        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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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
!= 1111	0	0	0	1	1	0	0	0	Rn	Rd	(1)	(1)	1	1	1	0	0	1	Rt
cond																			

A1

```
STREX{<c>}{<q>} <Rd>, <Rt>, [<Rn> {, {#}<imm>}]
```

```
d = UInt(Rd);  t = UInt(Rt);  n = UInt(Rn);  imm32 = Zeros(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	1	0	1	0	0	0	0	1	0	0	Rn	Rt	Rd	imm8

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 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.

<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', 32);
    else
        R[d] = ZeroExtend('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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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

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

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	0	Rd

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', 32);
    else
        R[d] = ZeroExtend('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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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;
```

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	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				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;
```

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.

<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(AccessType_GPR) then R[t]:R[t2] else R[t2]:R[t];

    if AArch32.ExclusiveMonitorsPass(address,8) then
        MemA[address,8] = value;
        R[d] = ZeroExtend('0', 32);
    else
        R[d] = ZeroExtend('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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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

31-28	27	26	25	24	23	22	21	20	19-16	15-12	11	10	9	8	7	6	5	4	3-0
!= 1111	0	0	0	1	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	3-0	15-12	11	10	9	8	7	6	5	4	3-0
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

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n];
    if AArch32.ExclusiveMonitorsPass(address,2) then
        MemA[address,2] = R[t]<15:0>;
        R[d] = ZeroExtend('0', 32);
    else
        R[d] = ZeroExtend('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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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 == 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
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

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	0	1	0	1	0	!= 1111	Rt	imm12
												Rn		

T2

```
STRH{<c>}.W <Rt>, [<Rn> {, #{+}<imm>}] // (<Rt>, <Rn>, <imm> can be represented in T1)
```

```
STRH{<c>}{<q>} <Rt>, [<Rn> {, #{+}<imm>}]
```

```
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
```

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	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	1	0	!= 1111	Rt	1	P	U	W	imm8
												Rn						

Offset (P == 1 && U == 0 && W == 0)

```
STRH{<c>}{<q>} <Rt>, [<Rn> {, #-<imm>}]
```

Post-indexed (P == 0 && W == 1)

```
STRH{<c>}{<q>} <Rt>, [<Rn>], #{+/-}<imm>
```

Pre-indexed (P == 1 && W == 1)

```
STRH{<c>}{<q>} <Rt>, [<Rn>, #{+/-}<imm>]!
```

```
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
```

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.

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”:

U	+/-
0	-
1	+

\+ 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 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

```
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,2] = R[t]<15:0>;
        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,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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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) = (SRType_LSL, 0);
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	0	1	Rm	Rn	Rt

T1

```
STRH{<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);
```

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 0 0	0 1	0	!= 1111	Rt	0 0 0 0 0 0	imm2	Rm
			Rn				

T2

```
STRH{<c>}.W <Rt>, [<Rn>, {+}<Rm>] // (<Rt>, <Rn>, <Rm> can be represented in T1)

STRH{<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) = (SRType_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”:

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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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

```
STRHT{<c>}{<q>} <Rt>, [<Rn>], {+/-}<Rm>
```

```
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 `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

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	0	0	0	1	0	!= 1111				Rt				1	1	1	0	imm8							
												Rn																			

T1

```
STRHT{<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*.

<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”:

U	+/-
0	-
1	+

For encoding A2: specifies the index register is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

U	+/-
0	-
1	+

<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
    EncodingSpecificOperations();
    if PSTATE.EL == EL2 then UNPREDICTABLE;               // Hyp mode
    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_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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

31-28	27	26	25	24	23	22	21	20	19-16	15-12	11-0
!= 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');
register_form = FALSE;  imm32 = ZeroExtend(imm12, 32);
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

31-28	27	26	25	24	23	22	21	20	19-16	15-12	11-7	6-5	4	3-0
!= 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');
register_form = TRUE;  (shift_t, shift_n) = DecodeImmShift(stype, imm5);
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 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	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	!= 1111	Rt	1	1	1	0	imm8
												Rn						

T1

```
STRT{<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*.

<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”:

U	+/-
0	-
1	+

For encoding A2: specifies the index register is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

U	+/-
0	-
1	+

<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
    EncodingSpecificOperations();
    if PSTATE.EL == EL2 then UNPREDICTABLE;                // Hyp mode
    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;
    bits(32) data;
    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;
```

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 `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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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, any CONSTRAINED UNPREDICTABLE behavior, and any operational information 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	0	1 1 1 1	Rd	imm12
cond						

A2

```
SUB{<c>}{<q>} <Rd>, PC, #<const>
```

is equivalent to

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

and is the preferred disassembly when `imm12 == '000000000000'`.

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 0	i	1 0	1	0	1	0	1 1 1 1	0	imm3	Rd	imm8

T2

```
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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11–0
!= 1111	0	0	1	0	0	1	0	S	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	8–6	5–3	2–0
0	0	0	1	1	1	1	imm3	Rn	Rd

T1

```
SUB<c>{<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	10–8	7–0
0	0	1	1	1	Rdn	imm8

T2

```
SUB<c>{<q>} <Rdn>, #<imm8> // (Inside IT block, and <Rdn>, <imm8> can be represented in T1)

SUB<c>{<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 = UInt(Rdn);  n = UInt(Rdn);  setflags = !InITBlock();  imm32 = ZeroExtend(imm8, 32);
```

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 0	i	0	1 1 0 1	S	!= 1101 (Rn)	0	imm3	Rd	imm8

SUB (S == 0)

```
SUB<c>.W {<Rd>,} <Rn>, #<const> // (Inside IT block, and <Rd>, <Rn>, <const> can be represented in T1 or

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

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

```
SUBS.W {<Rd>,} <Rn>, #<const> // (Outside IT block, and <Rd>, <Rn>, <const> can be represented in T1 or T

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

```
if Rd == '1111' && S == '1' then SEE "CMP (immediate)";
if Rn == '1101' then SEE "SUB (SP minus immediate)";
d = UInt(Rd);  n = UInt(Rn);  setflags = (S == '1');  imm32 = T32ExpandImm(i:imm3:imm8);
if (d == 15 && !setflags) || n == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

T4

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	1 0	1	0	1	0	!= 11x1 (Rn)	0	imm3	Rd	imm8

T4

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

SUBW{<c>}{<q>} {<Rd>,} <Rn>, #<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);  n = UInt(Rn);  setflags = FALSE;  imm32 = ZeroExtend(i:imm3:imm8, 32);
if d == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

T5

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 0 1	Rn	1 0	(0)	0	(1)	(1)	(1)	(1)	imm8

T5 (!(Rn == 1110 && imm8 == 00000000))

```
SUBS{<c>}{<q>} PC, LR, #<imm8>
```

```
if Rn == '1110' && IsZero(imm8) then SEE "ERET";
d = 15;  n = UInt(Rn);  setflags = TRUE;  imm32 = ZeroExtend(imm8, 32);
if n != 14 then UNPREDICTABLE;
if InITBlock() && !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.

For encoding T5: is a 8-bit unsigned immediate, in the range 0 to 255, encoded in the "imm8" field. If <Rn> is the LR, and zero is used, see *ERET*.

<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.

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>.

<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*.

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 *SUB (SP minus 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.

In the T32 instruction set, MOVS{<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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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	S	!= 1101	Rd	imm5	stype	0	Rm
cond									Rn					

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

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

T1

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

```
SUBS{<q>} {<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

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	1	1	0	1	S	!= 1101	(0)	imm3	Rd	imm2	stype	Rm
												Rn						

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

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

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

```
SUB<c>.W {<Rd>,} <Rn>, <Rm> // (Inside IT block, and <Rd>, <Rn>, <Rm> can be represented in T1)

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

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

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

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

```
SUBS.W {<Rd>,} <Rn>, <Rm> // (Outside IT block, and <Rd>, <Rn>, <Rm> can be represented in T1)

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)";
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> 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 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 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 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 *SUB (SP minus register)*.

For encoding T1: 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 "Rn" field. If the SP is used, see *SUB (SP minus 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 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”:

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.

Operation

```
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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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”:

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 = 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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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

31-28	27	26	25	24	23	22	21	20	19	18	17	16	15-12	11-0
!= 1111	0	0	1	0	0	1	0	S	1	1	0	1	Rd	imm12
cond														

SUB (S == 0)

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

SUBS (S == 1)

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

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

T1

15	14	13	12	11	10	9	8	7	6-0
1	0	1	1	0	0	0	0	1	imm7

T1

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

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

T2

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

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	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	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(R[13], 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;
```

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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	1	0	S	1	1	0	1	Rd	imm5	stype	0	Rm
cond																	

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

```
SUB{<c>}{<q>} {<Rd>,} SP, <Rm> , RRX
```

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

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

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

```
SUBS{<c>}{<q>} {<Rd>,} SP, <Rm> , RRX
```

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

```
SUBS{<c>}{<q>} {<Rd>,} SP, <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	15	14 13 12	11 10 9 8	7 6	5 4	3 2 1 0
1	1	1	0	1	0	1	1	1	0	1	S	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>}.W {<Rd>,} SP, <Rm> // (<Rd>, <Rm> can be represented in T1 or T2)
```

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

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>}
```

```
if Rd == '1111' && S == '1' then SEE "CMP (register)";
d = UInt(Rd);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(stype, imm3:imm2);
if (d == 15 && !setflags) || 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> 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”:

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 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

```
if ConditionPassed() then
    EncodingSpecificOperations();
    shifted = Shift(R[m], shift_t, shift_n, PSTATE.C);
    (result, nzcv) = AddWithCarry(R[13], 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;
```

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

SVC

Supervisor Call causes a Supervisor Call exception. For more information, see *Supervisor Call (SVC) exception*.

Note

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

31 30 29 28	27	26	25	24	23 ... 0
!= 1111	1	1	1	1	imm24
cond					

A1

```
SVC{<c>}{<q>} {#}<imm>
```

```
imm32 = ZeroExtend(imm24, 32);
```

T1

15	14	13	12	11	10	9	8	7 ... 0
1	1	0	1	1	1	1	1	imm8

T1

```
SVC{<c>}{<q>} {#}<imm>
```

```
imm32 = ZeroExtend(imm8, 32);
```

Assembler Symbols

<c> See *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<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>);
```

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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

31-28	27-23	22	21-20	19-16	15-12	11-10	9	8	7-4	3-0
!= 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

15-7	6-5	4	3-0	15-12	11-8	7	6	5-4	3-0
1 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”:

rotate	<amount>
00	(omitted)
01	8
10	16
11	24

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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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 0	!= 1111	Rd	rotate	(0)	(0)	0 1 1 1	Rm
cond				Rn						

A1

```
SXTAB16{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, ROR #<amount>}
```

```
if Rn == '1111' then SEE "SXTB16";
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	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	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";
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”:

rotate	<amount>
00	(omitted)
01	8
10	16
11	24

Operation

```
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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

SXTAH

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

31 30 29 28	27	26	25	24	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	!= 1111	Rd	rotate	(0)	(0)	0	1	1	1	Rm
cond									Rn									

A1

```
SXTAH{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, ROR #<amount>}
```

```
if Rn == '1111' then SEE "SXTH";
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	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	0	!= 1111	1	1	1	1	Rd	1	(0)	rotate	Rm
												Rn									

T1

```
SXTAH{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, ROR #<amount>}
```

```
if Rn == '1111' then SEE "SXTH";
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”:

rotate	<amount>
00	(omitted)
01	8
10	16
11	24

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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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

31-28	27	26	25	24	23	22	21	20	19	18	17	16	15-12	11-10	9	8	7	6	5	4	3-0
!= 1111	0	1	1	0	1	0	1	0	1	1	1	1	Rd	rotate	(0)	(0)	0	1	1	1	Rm
cond																					

A1

```
SXTB{<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	5-3	2-0
1	0	1	1	0	0	1	0	0	1	Rm	Rd

T1

```
SXTB{<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	15	14	13	12	11-8	7	6	5-4	3-0
1	1	1	1	1	0	1	0	0	1	0	0	1	1	1	1	1	1	1	1	Rd	1	(0)	rotate	Rm

T2

```
SXTB{<c>}.W {<Rd>,} <Rm> // (<Rd>, <Rm> can be represented in T1)
```

```
SXTB{<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”:

rotate	<amount>
00	(omitted)
01	8
10	16
11	24

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    rotated = ROR(R[m], rotation);
    R[d] = 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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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

31 30 29 28	27 26 25 24 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 0	1 1 1 1	Rd	rotate	(0)	(0)	0 1 1 1	Rm
cond										

A1

```
SXTB16{<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	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	0	1 1 1 1	1 1 1 1	Rd	1	(0)	rotate	Rm

T1

```
SXTB16{<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”:

rotate	<amount>
00	(omitted)
01	8
10	16
11	24

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    rotated = ROR(R[m], rotation);
    R[d]<15:0>  = SignExtend(rotated<7:0>, 16);
    R[d]<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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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)	(0)	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	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	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	0	1 1 1 1	1 1 1 1	Rd	1	(0)	rotate	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”:

rotate	<amount>
00	(omitted)
01	8
10	16
11	24

Operation

```
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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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

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				(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 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*.

<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();
    integer halfwords;
    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);
```

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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 1 0	0 1	1	Rn	(0)	(0)	(0)	(0)	imm12
cond									

A1

```
TEQ{<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	15	14 13 12	11 10 9 8	7 6 5 4 3 2 1 0
1 1 1 1 0	i	0	0 1 0 0	1	Rn	0	imm3	1 1 1 1	imm8

T1

```
TEQ{<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] 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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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	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	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

<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: 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: 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

<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.

Operation

```
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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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

31 30 29 28	27 26 25 24 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	1	Rn	(0)	(0)	(0)	(0)	Rs	0	stype	1	Rm
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”:

stype	<type>
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] 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, see *Trace Synchronization Buffer (TSB CSYNC)*.

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
(FEAT_TRF)

31-28	27	26	25	24	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	1	0	0	0	0	0	(1)	(1)	(1)	(1)	(0)	(0)	(0)	(0)	0	0	0	1	0	0	1	0
cond																												

A1

```
TSB{<c>}{<q>} CSYNC
```

```
if !HaveSelfHostedTrace() then EndOfInstruction(); // Instruction executes as NOP
if cond != '1110' then UNPREDICTABLE;              // TSB 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
(FEAT_TRF)

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	0	(1)	(1)	(1)	(1)	1	0	(0)	0	(0)	0	0	0	0	0	0	1	0	0	1	0

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	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
```

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_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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

31	30	29	28	27	26	25	24	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)	imm5					stype		0	Rm			
cond																															

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

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	0	Rm			Rn		

T1

```
TST{<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	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	0	0	1	Rn				(0)	imm3			1	1	1	1	imm2		stype		Rm			

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":

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.

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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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”:

stype	<type>
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;
    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

31-28	27-23	22-20	19-16	15-12	11	10	9	8	7	6	5	4	3-0
!= 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

15-7	6-4	3-0	15-12	11-8	7	6	5	4	3-0
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>
```

```
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]<15:0>) + UInt(R[m]<15:0>);
    sum2 = UInt(R[n]<31:16>) + UInt(R[m]<31:16>);
    R[d]<15:0>  = sum1<15:0>;
    R[d]<31:16> = sum2<15:0>;
    PSTATE.GE<1:0> = if sum1 >= 0x10000 then '11' else '00';
    PSTATE.GE<3:2> = if sum2 >= 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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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

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

A1

```
UADD8{<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	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	0	Rm

T1

```
UADD8{<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<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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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

31 30 29 28	27	26	25	24	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	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

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

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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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

31 30 29 28	27 26 25 24 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	1	1	widthm1	Rd	lsb	1 0 1	Rn
cond								

A1

```
UBFX{<c>}{<q>} <Rd>, <Rn>, #<lsb>, #<width>
```

```
d = UInt(Rd);  n = UInt(Rn);
lsbit = UInt(lsb);  widthminus1 = UInt(widthm1);
msbit = lsbit + widthminus1;
if d == 15 || n == 15 then UNPREDICTABLE;
if msbit > 31 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `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.

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	0	0	Rn	0	imm3	Rd	imm2	(0)	widthm1

T1

```
UBFX{<c>}{<q>} <Rd>, <Rn>, #<lsb>, #<width>
```

```
d = UInt(Rd);  n = UInt(Rn);
lsbit = UInt(imm3:imm2);  widthminus1 = UInt(widthm1);
msbit = lsbit + widthminus1;
if d == 15 || n == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
if msbit > 31 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `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.

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

```
if ConditionPassed() then
    EncodingSpecificOperations();
    R[d] = ZeroExtend(R[n]<msbit:lsbit>, 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_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

UDF

Permanently Undefined generates an Undefined Instruction exception.

The encodings for `UDF` used in this section are defined as permanently UNDEFINED. 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 1	imm12	1 1 1 1	imm4
cond				

A1

```
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 0 1 1 1 1	0	imm8

T1

```
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	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	imm4	1 0	1	0	imm12

T2

```
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

```
if ConditionPassed() then
    EncodingSpecificOperations();
    UNDEFINED;
```

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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

31 30 29 28	27	26	25	24	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	1	1	Rd	(1) (1) (1) (1)	Rm	0	0	0	1	Rn
cond										Ra						

A1

```
UDIV{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  a = UInt(Ra);
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

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	1	1	Rn	(1) (1) (1) (1)	Rd	1	1	1	1	Rm
													Ra						

T1

```
UDIV{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  a = UInt(Ra);
// Armv8-A removes UNPREDICTABLE for R13
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.

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 dividend, encoded in the "Rn" field.

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

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    integer result;
    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_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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

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

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

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

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
    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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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

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>;
```

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_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

UHASX

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

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

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

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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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 = 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	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	1	1	0	Rm

T1

```
UHSAX{<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();
    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>  = 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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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
cond																			

A1

```
UHSUB16{<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	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	0	1	Rn	1	1	1	1	Rd	0	1	1	0	Rm

T1

```
UHSUB16{<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 = 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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

UHSUB8

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

31 30 29 28	27 26 25 24 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)	1	1 1	1	Rm
cond												

A1

```
UHSUB8{<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	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 0 0	Rn	1 1 1 1	Rd	0	1	1	0	Rm

T1

```
UHSUB8{<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 = 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>;
```

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_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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

31 30 29 28	27 26 25 24	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	0	RdHi	RdLo	Rm	1 0 0 1	Rn
cond								

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

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	1 1 0	Rn	RdLo	RdHi	0 1 1 0	Rm

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

```
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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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 = 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;
```

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	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	0	Rn				RdLo				RdHi				0	0	0	0	Rm			

T1

```
UMLAL{<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;
```

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 = 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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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

31 30 29 28	27 26 25 24	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	S	RdHi	RdLo	Rm	1 0 0 1	Rn
cond								

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;
```

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	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 1 0	Rn	RdLo	RdHi	0 0 0 0	Rm

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;
```

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 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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

UQADD16

Unsigned Saturating Add 16 performs two unsigned 16-bit integer additions, saturates the results to the 16-bit unsigned integer range 0 <= x <= 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																			

A1

```
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	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>	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]<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);
```

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

UQADD8

Unsigned Saturating Add 8 performs four unsigned 8-bit integer additions, saturates the results to the 8-bit unsigned integer range 0 <= x <= 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	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);
```

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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 <= x <= 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	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

```
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_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

UQSAX

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 $0 <= x <= 2^{16} - 1$, and writes the results to the destination register.

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

A1

31-28	27	26	25	24	23	22	21	20	19-16	15-12	11	10	9	8	7	6	5	4	3-0
!= 1111	0	1	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>
```

```
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	3-0	15	14	13	12	11-8	7	6	5	4	3-0
1	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>
```

```
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();
    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_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

UQSUB16

Unsigned Saturating Subtract 16 performs two unsigned 16-bit integer subtractions, saturates the results to the 16-bit unsigned integer range 0 <= x <= 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	1 1	1	Rm
cond												

A1

```
UQSUB16{<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	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 0 1	Rn	1 1 1 1	Rd	0	1	0	1	Rm

T1

```
UQSUB16{<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 = 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>  = UnsignedSat(diff1, 16);
    R[d]<31:16> = UnsignedSat(diff2, 16);
```

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

UQSUB8

Unsigned Saturating Subtract 8 performs four unsigned 8-bit integer subtractions, saturates the results to the 8-bit unsigned integer range 0 <= x <= 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	1 1	1	Rm
cond												

A1

```
UQSUB8{<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	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 0 0	Rn	1 1 1 1	Rd	0	1	0	1	Rm

T1

```
UQSUB8{<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 = 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>   = UnsignedSat(diff1, 8);
    R[d]<15:8>  = UnsignedSat(diff2, 8);
    R[d]<23:16> = UnsignedSat(diff3, 8);
    R[d]<31:24> = UnsignedSat(diff4, 8);
```

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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

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

A1

```
USAD8{<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	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>
```

```
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();
    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 = absdiff1 + absdiff2 + absdiff3 + absdiff4;
    R[d] = result<31:0>;
```

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

USADA8

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

31 30 29 28	27 26 25 24 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 0 0	Rd	!= 1111	Rm	0 0 0 1	Rn
cond			Ra			

A1

```
USADA8{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>
```

```
if Ra == '1111' then SEE "USAD8";
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  a = UInt(Ra);
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	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	!= 1111	Rd	0 0	0 0	Rm
			Ra				

T1

```
USADA8{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>
```

```
if Ra == '1111' then SEE "USAD8";
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  a = 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*.
<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.
<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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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

31	30	29	28	27	26	25	24	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	sat_imm					Rd				imm5					sh	0	1	Rn			
cond																															

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>}
```

```
d = UInt(Rd);  n = UInt(Rn);  saturate_to = UInt(sat_imm);
(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	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	sh	0	Rn				0	imm3			Rd				imm2		(0)	sat_imm				

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";
d = UInt(Rd);  n = UInt(Rn);  saturate_to = UInt(sat_imm);
(shift_t, shift_n) = DecodeImmShift(sh:'0', imm3: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 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

```
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';
```

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

USAT16

Unsigned Saturate 16 saturates two signed 16-bit values 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

31 30 29 28	27	26	25	24	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
cond																			

A1

```
USAT16{<c>}{<q>} <Rd>, #<imm>, <Rn>
```

```
d = UInt(Rd);  n = UInt(Rn);  saturate_to = UInt(sat_imm);
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	0	(0)	1	1	1	0	1	0	Rn	0	0	0	0	Rd	0	0	(0)	(0)	sat_imm

T1

```
USAT16{<c>}{<q>} <Rd>, #<imm>, <Rn>
```

```
d = UInt(Rd);  n = UInt(Rn);  saturate_to = UInt(sat_imm);
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 0 to 15, encoded in the "sat_imm" field.

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

Operation

```
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';
```

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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

31-28	27	26	25	24	23	22	21	20	19-16	15-12	11	10	9	8	7	6	5	4	3-0
!= 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 = 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	3-0	15	14	13	12	11-8	7	6	5	4	3-0
1	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 = 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();
    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>  = sum<15:0>;
    R[d]<31:16> = diff<15:0>;
    PSTATE.GE<1:0> = if sum  >= 0x10000 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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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

31 30 29 28	27 26 25 24 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 1	1	Rm
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

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

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

<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 = 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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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

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

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

```
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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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

31-28	27-23	22	21-20	19-16	15-12	11-10	9	8	7-4	3-0
!= 1111	0 1 1 0 1	1	1 0	!= 1111	Rd	rotate	(0)	(0)	0 1 1 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

15-7	6-5	4	3-0	15-12	11-8	7	6	5-4	3-0
1 1 1 1 1 0 1 0 0	1 0	1	!= 1111	1 1 1 1	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

<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”:

rotate	<amount>
00	(omitted)
01	8
10	16
11	24

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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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

31 30 29 28	27 26 25 24 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	0 0	!= 1111	Rd	rotate	(0)	(0)	0 1 1 1	Rm
cond				Rn						

A1

```
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;
```

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 0 0	0 1	1	!= 1111	1 1 1 1	Rd	1	(0)	rotate	Rm
			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”:

rotate	<amount>
00	(omitted)
01	8
10	16
11	24

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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

UXTAH

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

31 30 29 28	27 26 25 24 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	!= 1111	Rd	rotate	(0)	(0)	0 1 1 1	Rm
cond				Rn						

A1

```
UXTAH{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, ROR #<amount>}
```

```
if Rn == '1111' then SEE "UXTH";
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	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	!= 1111	1 1 1 1	Rd	1	(0)	rotate	Rm
			Rn						

T1

```
UXTAH{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, ROR #<amount>}
```

```
if Rn == '1111' then SEE "UXTH";
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”:

rotate	<amount>
00	(omitted)
01	8
10	16
11	24

Operation

```
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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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 1	Rd	rotate	(0)	(0)	0 1 1 1	Rm
cond										

A1

```
UXTB{<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	1	1	Rm	Rd

T1

```
UXTB{<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	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	1 1 1 1	Rd	1	(0)	rotate	Rm

T2

```
UXTB{<c>}.W {<Rd>,} <Rm> // (<Rd>, <Rm> can be represented in T1)
```

```
UXTB{<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”:

rotate	<amount>
00	(omitted)
01	8
10	16
11	24

Operation

```
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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

UXTB16

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

31 30 29 28	27 26 25 24 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	0 0	1 1 1 1	Rd	rotate	(0)	(0)	0 1 1 1	Rm
cond										

A1

```
UXTB16{<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	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	1 1 1 1	Rd	1	(0)	rotate	Rm

T1

```
UXTB16{<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> 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”:

rotate	<amount>
00	(omitted)
01	8
10	16
11	24

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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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	Rd	rotate	(0)	(0)	0 1 1 1	Rm
cond										

A1

```
UXTH{<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	1	0	Rm	Rd

T1

```
UXTH{<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	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 1	Rd	1	(0)	rotate	Rm

T2

```
UXTH{<c>}.W {<Rd>,} <Rm> // (<Rd>, <Rm> can be represented in T1)
```

```
UXTH{<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”:

rotate	<amount>
00	(omitted)
01	8
10	16
11	24

Operation

```
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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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

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

A1

```
WFE{<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	1	0	0	1	0	0	0	0	0

T1

```
WFE{<c>}{<q>}
```

```
// No additional decoding required
```

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	0	0	1	1	1	0	1	0	(1)	(1)	(1)	(1)	1	0	(0)	0	(0)	0	0	0	0	0	0	0	0	0	1	0

T2

```
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*.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if IsEventRegisterSet() then
        ClearEventRegister();
    else
        if PSTATE.EL == EL0 then
            // Check for traps described by the OS.
            AArch32.CheckForWFxTrap(EL1, WFxType_WFE);
        if PSTATE.EL IN {EL0, EL1} && EL2Enabled() && !IsInHost() then
            // Check for traps described by the Hypervisor.
            AArch32.CheckForWFxTrap(EL2, WFxType_WFE);
        if HaveEL(EL3) && PSTATE.M != M32_Monitor then
            // Check for traps described by the Secure Monitor.
            AArch32.CheckForWFxTrap(EL3, WFxType_WFE);
        integer localtimeout = 1 << 64;  // No local timeout event is generated
        WaitForEvent(localtimeout);
```

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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

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

A1

```
WFI{<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	1	0	0	1	1	0	0	0	0

T1

```
WFI{<c>}{<q>}
```

```
// No additional decoding required
```

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	0	0	1	1	1	0	1	0	(1)	(1)	(1)	(1)	1	0	(0)	0	(0)	0	0	0	0	0	0	0	0	0	1	1

T2

```
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*.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if !InterruptPending() then
        if PSTATE.EL == EL0 then
            // Check for traps described by the OS.
            AArch32.CheckForWFxTrap(EL1, WFxType_WFI);
        if PSTATE.EL IN {EL0, EL1} && EL2Enabled() && !IsInHost() then
            // Check for traps described by the Hypervisor.
            AArch32.CheckForWFxTrap(EL2, WFxType_WFI);
        if HaveEL(EL3) && PSTATE.M != M32_Monitor then
            // Check for traps described by the Secure Monitor.
            AArch32.CheckForWFxTrap(EL3, WFxType_WFI);
        integer localtimeout = 1 << 64;  // No local timeout event is generated
        WaitForInterrupt(localtimeout);
```

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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

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

A1

```
YIELD{<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	1	0	0	0	1	0	0	0	0

T1

```
YIELD{<c>}{<q>}
```

```
// No additional decoding required
```

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	0	0	1	1	1	0	1	0	(1)	(1)	(1)	(1)	1	0	(0)	0	(0)	0	0	0	0	0	0	0	0	0	0	1

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

```
if ConditionPassed() then
    EncodingSpecificOperations();
    Hint_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, FSTMIAX: 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).

VADDHN: 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).

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.

VFMAB, VFMAT (BFloat16, by scalar): BFloat16 floating-point widening multiply-add long (by scalar).

VFMAB, VFMAT (BFloat16, vector): BFloat16 floating-point widening multiply-add long (vector).

VFMAL (by scalar): Vector Floating-point Multiply-Add Long to accumulator (by scalar).

VFMAL (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.

VJCVT: 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, VLDMDB, VLDMIA: 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.

VQDMLSL: Vector Saturating Doubling Multiply Subtract Long.

VQDMULH: Vector Saturating Doubling Multiply Returning High Half.

VQDMULL: Vector Saturating Doubling Multiply Long.

VQMOVN, VQMOVUN: Vector Saturating Move and Narrow.

VQNEG: Vector Saturating Negate.

VQRDMLAH: Vector Saturating Rounding Doubling Multiply Accumulate Returning High Half.

VQRDMLSH: Vector Saturating Rounding Doubling Multiply Subtract Returning High Half.

VQRDMULH: Vector Saturating Rounding Doubling Multiply Returning High Half.

VQRSHL: Vector Saturating Rounding Shift Left.

VQRSHRN (zero): Vector Saturating Rounding Shift Right, Narrow: an alias of VQMOVN, VQMOVUN.

VQRSHRN, VQRSHRUN: Vector Saturating Rounding Shift Right, Narrow.

VQRSHRUN (zero): Vector Saturating Rounding Shift Right, Narrow: an alias of VQMOVN, VQMOVUN.

VQSHL (register): Vector Saturating Shift Left (register).

VQSHL, VQSHLU (immediate): Vector Saturating Shift Left (immediate).

VQSHRN (zero): Vector Saturating Shift Right, Narrow: an alias of VQMOVN, VQMOVUN.

VQSHRN, VQSHRUN: Vector Saturating Shift Right, Narrow.

VQSHRUN (zero): Vector Saturating Shift Right, Narrow: an alias of VQMOVN, VQMOVUN.

VQSUB: Vector Saturating Subtract.

VRADDHN: Vector Rounding Add and Narrow, returning High Half.

VRECPE: Vector Reciprocal Estimate.

VRECPS: Vector Reciprocal Step.

VREV16: Vector Reverse in halfwords.

VREV32: Vector Reverse in words.

VREV64: Vector Reverse in doublewords.

VRHADD: Vector Rounding Halving Add.

VRINTA (Advanced SIMD): Vector Round floating-point to integer towards Nearest with Ties to Away.

VRINTA (floating-point): Round floating-point to integer to Nearest with Ties to Away.

VRINTM (Advanced SIMD): Vector Round floating-point to integer towards -Infinity.

VRINTM (floating-point): Round floating-point to integer towards -Infinity.

VRINTN (Advanced SIMD): Vector Round floating-point to integer to Nearest.

VRINTN (floating-point): Round floating-point to integer to Nearest.

VRINTP (Advanced SIMD): Vector Round floating-point to integer towards +Infinity.

VRINTP (floating-point): Round floating-point to integer towards +Infinity.

VRINTR: Round floating-point to integer.

VRINTX (Advanced SIMD): Vector round floating-point to integer inexact.

VRINTX (floating-point): Round floating-point to integer inexact.

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.

VRSQRTE: Vector Reciprocal Square Root Estimate.

VRSQRTS: 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.

VSMMLA: Widening 8-bit signed integer matrix multiply-accumulate into 2x2 matrix.

VSQRT: 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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

AESD

AES single round decryption.

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

A1
(FEAT_AES)

31	30	29	28	27	26	25	24	23	22	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	1	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
(FEAT_AES)

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		0	0	Vd				0	0	1	1	0	1	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”:

size	<dt>
00	8
01	RESERVED
1x	RESERVED

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

AESE

AES single round encryption.

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

A1
(FEAT_AES)

31	30	29	28	27	26	25	24	23	22	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
(FEAT_AES)

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		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”:

size	<dt>
00	8
01	RESERVED
1x	RESERVED

<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(AESShiftRows(op1 EOR op2));
```

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_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

AESIMC

AES inverse mix columns.

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

A1
(FEAT_AES)

31	30	29	28	27	26	25	24	23	22	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

A1

```
AESIMC.<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
(FEAT_AES)

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	0	0	Vd	0	0	1	1	1	1	M	0	Vm

T1

```
AESIMC.<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);
```

Assembler Symbols

<dt> Is the data type, encoded in “size”:

size	<dt>
00	8
01	RESERVED
1x	RESERVED

<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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

AESMC

AES mix columns.

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

A1
(FEAT_AES)

31	30	29	28	27	26	25	24	23	22	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
(FEAT_AES)

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		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”:

size	<dt>
00	8
01	RESERVED
1x	RESERVED

<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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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 *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 0	P	U	D	W	1	Rn	Vd	1 0	1 1	imm8<7:1>	1
cond												imm8<0>

Decrement Before (P == 1 && U == 0 && W == 1)

```
FLDMDBX{<c>}{<q>} <Rn>!, <dreglist>
```

Increment After (P == 0 && U == 1)

```
FLDMIAX{<c>}{<q>} <Rn>{!}, <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 `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 `regs > 16 || (d+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.

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	P	U	D	W	1	Rn	Vd	1 0	1 1	imm8<7:1>	1
											imm8<0>

Decrement Before (P == 1 && U == 0 && W == 1)

```
FLDMDBX{<c>}{<q>} <Rn>!, <dreglist>
```

Increment After (P == 0 && U == 1)

```
FLDMIAX{<c>}{<q>} <Rn>{!}, <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 `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 `regs > 16 || (d+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*.
<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.
<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 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(AccessType_ASIMD) then word1:word2 else word2:word1;

    if wback then R[n] = if add then R[n]+imm32 else R[n]-imm32;
```

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_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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

31 30 29 28	27	26	25	24	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>	1
cond																imm8<0>

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

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	P	U	D	W	0	Rn	Vd	1	0	1	1	imm8<7:1>	1
																			imm8<0>

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.

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 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.
            if BigEndian(AccessType_ASIMD) then
                MemA[address,4] =  D[d+r]<63:32>;
                MemA[address+4,4] =  D[d+r]<31:0>;
            else
                MemA[address,4] = D[d+r]<31:0>;
                MemA[address+4,4] = D[d+r]<63:32>;

            address = address+8;

    if wback then R[n] = if add then R[n]+imm32 else R[n]-imm32;
```

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_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

SHA1C

SHA1 hash update (choose).
It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1
(FEAT_SHA1)

31	30	29	28	27	26	25	24	23	22	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				1	1	0	0	N	Q	M	0	Vm			

A1

```
SHA1C.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
(FEAT_SHA1)

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	0	0	Vn				Vd				1	1	0	0	N	Q	M	0	Vm			

T1

```
SHA1C.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();
    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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

SHA1H

SHA1 fixed rotate.

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

A1
(FEAT_SHA1)

31	30	29	28	27	26	25	24	23	22	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 = UInt(D:Vd); m = UInt(M:Vm);
```

T1
(FEAT_SHA1)

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		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 = 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();
    Q[d>>1] = ZeroExtend(ROL(Q[m>>1]<31:0>, 30), 128);
```

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.

SHA1M

SHA1 hash update (majority).
It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1
(FEAT_SHA1)

31	30	29	28	27	26	25	24	23	22	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
(FEAT_SHA1)

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	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);
```

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]<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, 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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

SHA1P

SHA1 hash update (parity).
It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1
(FEAT_SHA1)

31	30	29	28	27	26	25	24	23	22	21	20	19-16	15-12	11	10	9	8	7	6	5	4	3-0
1	1	1	1	0	0	1	0	0	D	0	1	Vn	Vd	1	1	0	0	N	Q	M	0	Vm

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
(FEAT_SHA1)

15	14	13	12	11	10	9	8	7	6	5	4	3-0	15-12	11	10	9	8	7	6	5	4	3-0
1	1	1	0	1	1	1	1	0	D	0	1	Vn	Vd	1	1	0	0	N	Q	M	0	Vm

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);
```

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]<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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

SHA1SU0

SHA1 schedule update 0.
It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1
(FEAT_SHA1)

31	30	29	28	27	26	25	24	23	22	21	20	19-16	15-12	11	10	9	8	7	6	5	4	3-0
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
(FEAT_SHA1)

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

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

SHA1SU1

SHA1 schedule update 1.
It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1
(FEAT_SHA1)

31	30	29	28	27	26	25	24	23	22	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	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
(FEAT_SHA1)

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	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);
    Q[d>>1] = W3:W2:W1:W0;
```

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_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

SHA256H

SHA256 hash update part 1.
It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1
(FEAT_SHA256)

31	30	29	28	27	26	25	24	23	22	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	0	Vn				Vd				1	1	0	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
(FEAT_SHA256)

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	0	Vn				Vd				1	1	0	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
(FEAT_SHA256)

31	30	29	28	27	26	25	24	23	22	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	1	Vn				Vd				1	1	0	0	N	Q	M	0	Vm			

A1

```
SHA256H2.32 <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
(FEAT_SHA256)

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	1	Vn				Vd				1	1	0	0	N	Q	M	0	Vm			

T1

```
SHA256H2.32 <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[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
(FEAT_SHA256)

31	30	29	28	27	26	25	24	23	22	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
(FEAT_SHA256)

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	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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

SHA256SU1

SHA256 schedule update 1.
It has encodings from the following instruction sets: A32 (A1) and T32 (T1) .

A1
(FEAT_SHA256)

31	30	29	28	27	26	25	24	23	22	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>
```

```
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
(FEAT_SHA256)

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	1	0	Vn				Vd				1	1	0	0	N	Q	M	0	Vm			

T1

```
SHA256SU1.32 <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();
    bits(32) elt;
    bits(128) result;
    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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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
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> <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	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	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> <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”:

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.

<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 = 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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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											

A1

```
VABAL{<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	D	!= 11	Vn	Vd	0	1	0	1	N	0	M	0	Vm
										size											

T1

```
VABAL{<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

```
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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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	19-16	15-12	11	10	9	8	7	6	5	4	3-0
1	1	1	1	0	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;
integer esize;
integer elements;
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	3-0	15-12	11	10	9	8	7	6	5	4	3-0
1	1	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;
integer esize;
integer elements;
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”:

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.

Operation

```
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];
            Elem[D[d+r],e,esize] = FPAbs(FPSub(op1,op2,StandardFPSCRValue()));
```

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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
1	1	1	1	0	0	1	U	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;
```

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	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":

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.

<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 = 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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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	U	1	D	!= 11	Vn	Vd	0	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	D	!= 11	Vn	Vd	0	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

```
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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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	15-12	11	10	9	8	7	6	5	4	3-0
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> <Qd>, <Qm>
```

```
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

31-28	27	26	25	24	23	22	21	20	19	18	17	16	15-12	11	10	9-8	7	6	5	4	3-0
!= 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)
(FEAT_FP16)

```
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;
integer esize;
integer d;
integer m;
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	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	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> <Qd>, <Qm>
```

```
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	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	0	0	0	Vd				1	0	size		1	1	M	0	Vm			

Half-precision scalar (size == 01)
(FEAT_FP16)

```
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;
integer esize;
integer d;
integer m;
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

<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”:

F	size	<dt>
0	00	S8
0	01	S16
0	10	S32
1	01	F16
1	10	F32

<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

```
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
                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             // VFP instruction
        case esize of
            when 16 S[d] = Zeros(16) : FPAbs(S[m]<15:0>);
            when 32 S[d] = FPAbs(S[m]);
            when 64 D[d] = FPAbs(D[m]);
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check and is operating only on integer vector elements, then the following apply:

- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
- The response of this instruction to 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_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

VACGE

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

31 30 29 28 27 26 25	24	23	22	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
				op									

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>
```

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
or_equal = (op == '0');
integer esize;
integer elements;
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 1 1 0	D	0	sz	Vn	Vd	1 1 1 0	N	Q	M	1	Vm
				op									

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>
```

```
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');
integer esize;
integer elements;
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”:

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.

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]);
            boolean test_passed;
            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);
```

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 VACLT.

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	19-16	15-12	11	10	9	8	7	6	5	4	3-0
1	1	1	1	0	0	1	1	0	D	1	sz	Vn	Vd	1	1	1	0	N	Q	M	1	Vm
										op												

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>
```

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
or_equal = (op == '0');
integer esize;
integer elements;
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	3-0	15-12	11	10	9	8	7	6	5	4	3-0
1	1	1	1	1	1	1	1	0	D	1	sz	Vn	Vd	1	1	1	0	N	Q	M	1	Vm
										op												

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>
```

```
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');
integer esize;
integer elements;
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”:

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.

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]);
            boolean test_passed;
            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);
```

VACLE

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 VACGE. This means:

- The encodings in this description are named to match the encodings of VACGE.
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of VACGE gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information 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	19-16	15-12	11	10	9	8	7	6	5	4	3-0
1	1	1	1	0	0	1	1	0	D	0	sz	Vn	Vd	1	1	1	0	N	Q	M	1	Vm
										op												

64-bit SIMD vector (Q == 0)

```
VACLE{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

is equivalent to

```
VACGE{<c>}{<q>}.<dt> <Dd>, <Dm>, <Dn>
```

128-bit SIMD vector (Q == 1)

```
VACLE{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

is equivalent to

```
VACGE{<c>}{<q>}.<dt> <Qd>, <Qm>, <Qn>
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3-0	15-12	11	10	9	8	7	6	5	4	3-0
1	1	1	1	1	1	1	1	0	D	0	sz	Vn	Vd	1	1	1	0	N	Q	M	1	Vm
										op												

64-bit SIMD vector (Q == 0)

```
VACLE{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

is equivalent to

```
VACGE{<c>}{<q>}.<dt> <Dd>, <Dm>, <Dn>
```

128-bit SIMD vector (Q == 1)

```
VACLE{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

is equivalent to

```
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.

<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: 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”:

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.
<Dd>	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.

Operational information

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, any CONSTRAINED UNPREDICTABLE behavior, and any operational information 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	0	D	1	sz	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>
```

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	1	sz	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.

<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: 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”:

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.

<Dd> 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.

Operational information

The description of *VACGT* gives the operational pseudocode for this instruction.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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

31	30	29	28	27	26	25	24	23	22	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	0	1	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;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
advsimd = TRUE;
integer esize;
integer elements;
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	0	M	0	Vm
cond																		

Half-precision scalar (size == 01)
(FEAT_FP16)

```
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;
integer esize;
integer d;
integer n;
integer m;
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	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	sz	Vn				Vd				1	1	0	1	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;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
if sz == '1' && InITBlock() then UNPREDICTABLE;
advsimd = TRUE;
integer esize;
integer elements;
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	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	D	1	1	Vn				Vd				1	0	size		N	0	M	0	Vm			

Half-precision scalar (size == 01)
(FEAT_FP16)

```
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' && InITBlock()  then UNPREDICTABLE;
advsimd = FALSE;
integer esize;
integer d;
integer n;
integer m;
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”:

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
                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

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	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;
```

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”:

size	<dt>
00	I8
01	I16
10	I32
11	I64

<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
            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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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 *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	0	0	N	0	M	0	Vm
										size											

A1

```
VADDHN{<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	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	0	0	N	0	M	0	Vm
										size											

T1

```
VADDHN{<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”:

size	<dt>
00	I16
01	I32
10	I64

<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 = 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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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					

A1

```
VADDL{<c>}{<q>}.<dt> <Qd>, <Dn>, <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	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					

T1

```
VADDL{<c>}{<q>}.<dt> <Qd>, <Dn>, <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”:

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

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for e = 0 to elements-1
        integer op1;
        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_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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	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”:

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.

<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

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for e = 0 to elements-1
        integer op1;
        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_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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, any CONSTRAINED UNPREDICTABLE behavior, and any operational information 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	i	1	D	0	0	0	imm3			Vd				0	x	x	1	0	Q	1	1	imm4			
																				cmode											

64-bit SIMD vector (Q == 0)

```
VAND{<c>}{<q>}.I16 {<Dd>,} <Dd>, #<imm>
```

is equivalent to

```
VBIC{<c>}{<q>}.I16 <Dd>, #~<imm>
```

128-bit SIMD vector (Q == 1)

```
VAND{<c>}{<q>}.I16 {<Qd>,} <Qd>, #<imm>
```

is equivalent to

```
VBIC{<c>}{<q>}.I16 <Qd>, #~<imm>
```

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	i	1	D	0	0	0	imm3			Vd				1	0	x	1	0	Q	1	1	imm4			
																				cmode											

64-bit SIMD vector (Q == 0)

```
VAND{<c>}{<q>}.I32 {<Dd>,} <Dd>, #<imm>
```

is equivalent to

```
VBIC{<c>}{<q>}.I32 <Dd>, #~<imm>
```

128-bit SIMD vector (Q == 1)

```
VAND{<c>}{<q>}.I32 {<Qd>,} <Qd>, #<imm>
```

is equivalent to

```
VBIC{<c>}{<q>}.I32 <Qd>, #~<imm>
```

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	i	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)

```
VAND{<c>}{<q>}.I16 {<Dd>,} <Dd>, #<imm>
```

is equivalent to

```
VBIC{<c>}{<q>}.I16 <Dd>, #~<imm>
```

128-bit SIMD vector (Q == 1)

```
VAND{<c>}{<q>}.I16 {<Qd>,} <Qd>, #<imm>
```

is equivalent to

```
VBIC{<c>}{<q>}.I16 <Qd>, #~<imm>
```

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	i	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)

```
VAND{<c>}{<q>}.I32 {<Dd>,} <Dd>, #<imm>
```

is equivalent to

```
VBIC{<c>}{<q>}.I32 <Dd>, #~<imm>
```

128-bit SIMD vector (Q == 1)

```
VAND{<c>}{<q>}.I32 {<Qd>,} <Qd>, #<imm>
```

is equivalent to

```
VBIC{<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 VBIC (immediate) gives the operational pseudocode for this instruction.

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	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

```
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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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
1	1	1	1	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 = AdvSIMDExpandImm('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
1	1	1	1	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 = AdvSIMDExpandImm('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
1	1	1	i	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 {<Qd>,} <Qd>, #<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	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	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 = 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.

VBIC (register)

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

31	30	29	28	27	26	25	24	23	22	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	1	Vn				Vd				0	0	0	1	N	Q	M	1	Vm			

64-bit SIMD vector (Q == 0)

```
VBIC{<c>}{<q>}{.<dt>} {<Dd>,} <Dn>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VBIC{<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	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	1	Vn				Vd				0	0	0	1	N	Q	M	1	Vm			

64-bit SIMD vector (Q == 0)

```
VBIC{<c>}{<q>}{.<dt>} {<Dd>,} <Dn>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VBIC{<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] 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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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	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	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

```
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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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

31-25	24	23	22	21-20	19-16	15-12	11-8	7	6	5	4	3-0
1 1 1 1 0 0 1	1	0	D	1 0	Vn	Vd	0 0 0 1	N	Q	M	1	Vm
				op								

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

15-13	12	11-7	6	5-4	3-0	15-12	11-8	7	6	5	4	3-0
1 1 1	1	1 1 1 1 0	D	1 0	Vn	Vd	0 0 0 1	N	Q	M	1	Vm
				op								

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

```
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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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 1 0 0 1	1	0	D	0 1	Vn	Vd	0 0 0 1	N	Q	M	1	Vm
				op								

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>
```

```
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	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 1	Vn	Vd	0 0 0 1	N	Q	M	1	Vm
				op								

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>
```

```
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

```
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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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
(FEAT_FCMA)

31	30	29	28	27	26	25	24	23	22	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	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)

```
VCADD{<q>}.<dt> <Qd>, <Qn>, <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
(FEAT_FCMA)

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	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)

```
VCADD{<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”:

S	<dt>
0	F16
1	F32

<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”:

rot	<rotate>
0	90
1	270

Operation

```
EncodingSpecificOperations();
CheckAdvSIMDEnabled();
for r = 0 to regs-1
    operand1 = D[n+r];
    operand2 = D[m+r];
    for e = 0 to (elements DIV 2)-1
        bits(esize) element1;
        bits(esize) element3;
        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,esize],element1,StandardFPSCRValue());
        result2 = FPAdd(Elem[operand1,e*2+1,esize],element3,StandardFPSCRValue());
        Elem[D[d+r],e*2,esize] = result1;
        Elem[D[d+r],e*2+1,esize] = result2;
```

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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

31	30	29	28	27	26	25	24	23	22	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	0	1	0	Q	M	0	Vm			

64-bit SIMD vector (Q == 0)

```
VCEQ{<c>}{<q>}.<dt> {<Dd>,} <Dm>, #0
```

128-bit SIMD vector (Q == 1)

```
VCEQ{<c>}{<q>}.<dt> {<Qd>,} <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;
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	1	D	1	1	size		0	1	Vd				0	F	0	1	0	Q	M	0	Vm			

64-bit SIMD vector (Q == 0)

```
VCEQ{<c>}{<q>}.<dt> {<Dd>,} <Dm>, #0
```

128-bit SIMD vector (Q == 1)

```
VCEQ{<c>}{<q>}.<dt> {<Qd>,} <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;
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”:

F	size	<dt>
0	00	I8
0	01	I16
0	10	I32
1	01	F16
1	10	F32

<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
            boolean test_passed;
            if floating_point then
                bits(esize) zero = FPZero('0', esize);
                test_passed = FPCompareEQ(Elem[D[m+r],e,esize], zero, StandardFPSCRValue());
            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);
```

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;
integer esize;
integer elements;
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	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;
```

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	0	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)

```
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;
integer esize;
integer elements;
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*.

<q> See *Standard assembler syntax fields*.

<dt> For encoding A1 and T1: is the data type for the elements of the vectors, encoded in “size”:

size	<dt>
00	I8
01	I16
10	I32

For encoding A2 and T2: 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.

Operation

```
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];
            boolean test_passed;
            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_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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

31	30	29	28	27	26	25	24	23	22	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	0	0	1	Q	M	0	Vm			

64-bit SIMD vector (Q == 0)

```
VCGE{<c>}{<q>}.<dt> {<Dd>,} <Dm>, #0
```

128-bit SIMD vector (Q == 1)

```
VCGE{<c>}{<q>}.<dt> {<Qd>,} <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;
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	1	D	1	1	size		0	1	Vd				0	F	0	0	1	Q	M	0	Vm			

64-bit SIMD vector (Q == 0)

```
VCGE{<c>}{<q>}.<dt> {<Dd>,} <Dm>, #0
```

128-bit SIMD vector (Q == 1)

```
VCGE{<c>}{<q>}.<dt> {<Qd>,} <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;
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”:

F	size	<dt>
0	00	S8
0	01	S16
0	10	S32
1	01	F16
1	10	F32

<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
            boolean test_passed;
            if floating_point then
                bits(esize) zero = FPZero('0', esize);
                test_passed = FPCompareGE(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 and is operating only on integer vector elements, then the following apply:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to 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_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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;
integer esize;
integer elements;
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	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)

```
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;
```

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)

```
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;
if sz == '1' && InITBlock() then UNPREDICTABLE;
vtype = VCGEType_fp;
integer esize;
integer elements;
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*.

<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”:

U	size	<dt>
0	00	S8
0	01	S16
0	10	S32
1	00	U8
1	01	U16
1	10	U32

For encoding A2 and T2: 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.

Operation

```
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];
            boolean test_passed;
            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 and is operating only on integer vector elements, then the following apply:

- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
- The response of this instruction to 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_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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 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

31	30	29	28	27	26	25	24	23	22	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	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> {<Qd>,} <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;
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	1	D	1	1	size	0	1	Vd	0	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> {<Qd>,} <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;
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”:

F	size	<dt>
0	00	S8
0	01	S16
0	10	S32
1	01	F16
1	10	F32

<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
            boolean test_passed;
            if floating_point then
                bits(esize) zero = FPZero('0', esize);
                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 and is operating only on integer vector elements, then the following apply:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to 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_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

VCGT (register)

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;
integer esize;
integer elements;
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	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

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	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;
if sz == '1' && InITBlock() then UNPREDICTABLE;
vtype = VCGTtype_fp;
integer esize;
integer elements;
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*.

<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":

U	size	<dt>
0	00	S8
0	01	S16
0	10	S32
1	00	U8
1	01	U16
1	10	U32

For encoding A2 and T2: 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.

Operation

```
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];
            boolean test_passed;
            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 and is operating only on integer vector elements, then the following apply:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to 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_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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

31	30	29	28	27	26	25	24	23	22	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	0	1	1	Q	M	0	Vm			

64-bit SIMD vector (Q == 0)

```
VCLE{<c>}{<q>}.<dt> {<Dd>,} <Dm>, #0
```

128-bit SIMD vector (Q == 1)

```
VCLE{<c>}{<q>}.<dt> {<Qd>,} <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;
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	1	D	1	1	size		0	1	Vd				0	F	0	1	1	Q	M	0	Vm			

64-bit SIMD vector (Q == 0)

```
VCLE{<c>}{<q>}.<dt> {<Dd>,} <Dm>, #0
```

128-bit SIMD vector (Q == 1)

```
VCLE{<c>}{<q>}.<dt> {<Qd>,} <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;
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”:

F	size	<dt>
0	00	S8
0	01	S16
0	10	S32
1	01	F16
1	10	F32

<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
            boolean test_passed;
            if floating_point then
                bits(esize) zero = FPZero('0', esize);
                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 and is operating only on integer vector elements, then the following apply:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to 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_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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, any CONSTRAINED UNPREDICTABLE behavior, and any operational information 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	0	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	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)

```
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>
```

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)

```
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>
```

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”:

U	size	<dt>
0	00	S8
0	01	S16
0	10	S32
1	00	U8
1	01	U16
1	10	U32

For encoding A2 and T2: 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.

<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.

Operational information

The description of *VCGE_r* gives the operational pseudocode for this instruction.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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> <Dd>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VCLS{<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	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> <Dd>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VCLS{<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”:

size	<dt>
00	S8
01	S16
10	S32
11	RESERVED

<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] = 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_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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

31	30	29	28	27	26	25	24	23	22	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	0	0	Q	M	0	Vm			

64-bit SIMD vector (Q == 0)

```
VCLT{<c>}{<q>}.<dt> {<Dd>,} <Dm>, #0
```

128-bit SIMD vector (Q == 1)

```
VCLT{<c>}{<q>}.<dt> {<Qd>,} <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;
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	1	D	1	1	size		0	1	Vd				0	F	1	0	0	Q	M	0	Vm			

64-bit SIMD vector (Q == 0)

```
VCLT{<c>}{<q>}.<dt> {<Dd>,} <Dm>, #0
```

128-bit SIMD vector (Q == 1)

```
VCLT{<c>}{<q>}.<dt> {<Qd>,} <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;
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”:

F	size	<dt>
0	00	S8
0	01	S16
0	10	S32
1	01	F16
1	10	F32

<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
            boolean test_passed;
            if floating_point then
                bits(esize) zero = FPZero('0', esize);
                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 and is operating only on integer vector elements, then the following apply:

- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
- The response of this instruction to 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_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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, any CONSTRAINED UNPREDICTABLE behavior, and any operational information 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	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	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”:

U	size	<dt>
0	00	S8
0	01	S16
0	10	S32
1	00	U8
1	01	U16
1	10	U32

For encoding A2 and T2: 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.

<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.

Operational information

The description of *VCGT_r* gives the operational pseudocode for this instruction.

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> <Dd>, <Dm>
```

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	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	1	Q	M	0	Vm			

64-bit SIMD vector (Q == 0)

```
VCLZ{<c>}{<q>}.<dt> <Dd>, <Dm>
```

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”:

size	<dt>
00	I8
01	I16
10	I32
11	RESERVED

<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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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
(FEAT_FCMA)

31	30	29	28	27	26	25	24 23	22	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	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 !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
(FEAT_FCMA)

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	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”:

S	<dt>
0	F16
1	F32

<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”:

rot	<rotate>
00	0
01	90
10	180
11	270

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
        bits(esize) element1;
        bits(esize) element2;
        bits(esize) element3;
        bits(esize) element4;
        case rot of
            when '00'
                element1 = Elem[operand2,e*2,esize];
                element2 = Elem[operand1,e*2,esize];
                element3 = Elem[operand2,e*2+1,esize];
                element4 = Elem[operand1,e*2,esize];
            when '01'
                element1 = FPNeg(Elem[operand2,e*2+1,esize]);
                element2 = Elem[operand1,e*2+1,esize];
                element3 = Elem[operand2,e*2,esize];
                element4 = Elem[operand1,e*2+1,esize];
            when '10'
                element1 = FPNeg(Elem[operand2,e*2,esize]);
                element2 = Elem[operand1,e*2,esize];
                element3 = FPNeg(Elem[operand2,e*2+1,esize]);
                element4 = Elem[operand1,e*2,esize];
            when '11'
                element1 = Elem[operand2,e*2+1,esize];
                element2 = Elem[operand1,e*2+1,esize];
                element3 = FPNeg(Elem[operand2,e*2,esize]);
                element4 = Elem[operand1,e*2+1,esize];
        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;
```

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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

(FEAT_FCMA)

31	30	29	28	27	26	25	24	23	22	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	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 !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);
```

T1

(FEAT_FCMA)

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”:

rot	<rotate>
00	0
01	90
10	180
11	270

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
        bits(esize) element1;
        bits(esize) element2;
        bits(esize) element3;
        bits(esize) element4;
        case rot of
            when '00'
                element1 = Elem[operand2,index*2,esize];
                element2 = Elem[operand1,e*2,esize];
                element3 = Elem[operand2,index*2+1,esize];
                element4 = Elem[operand1,e*2,esize];
            when '01'
                element1 = FPNeg(Elem[operand2,index*2+1,esize]);
                element2 = Elem[operand1,e*2+1,esize];
                element3 = Elem[operand2,index*2,esize];
                element4 = Elem[operand1,e*2+1,esize];
            when '10'
                element1 = FPNeg(Elem[operand2,index*2,esize]);
                element2 = Elem[operand1,e*2,esize];
                element3 = FPNeg(Elem[operand2,index*2+1,esize]);
                element4 = Elem[operand1,e*2,esize];
            when '11'
                element1 = Elem[operand2,index*2+1,esize];
                element2 = Elem[operand1,e*2+1,esize];
                element3 = FPNeg(Elem[operand2,index*2,esize]);
                element4 = Elem[operand1,e*2+1,esize];
        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;
```

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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

31 30 29 28	27	26	25	24	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	0	1	M	0	Vm
cond																	E				

Half-precision scalar (size == 01)
(FEAT_FP16)

```
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' && cond != '1110' then UNPREDICTABLE;
quiet_nan_exc = (E == '1');  with_zero = FALSE;
integer esize;
integer d;
integer m;
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.

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	0	1	(0)	0	(0)	(0)	(0)	(0)
cond																	E							

Half-precision scalar (size == 01)

(FEAT_FP16)

```
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;
integer esize;
integer d;
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

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	1	D	1	1	0	1	0	0	Vd				1	0	size		0	1	M	0	Vm			
																								E							

Half-precision scalar (size == 01)

(FEAT_FP16)

```
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;
integer esize;
integer d;
integer m;
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.

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	0	1	1	1	0	1	D	1	1	0	1	0	1	Vd				1	0	size		0	1	(0)	0	(0)	(0)	(0)	(0)
																								E							

Half-precision scalar (size == 01)
(FEAT_FP16)

```
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;
integer esize;
integer d;
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' && 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', 16) 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', 32) else S[m];
            nzcv = FPCompare(S[d], op32, quiet_nan_exc, FPSCR[]);
        when 64
            bits(64) op64 = if with_zero then FPZero('0', 64) else D[m];
            nzcv = FPCompare(D[d], op64, quiet_nan_exc, FPSCR[]);

    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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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 *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
!= 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)
(FEAT_FP16)

```
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;
integer esize;
integer d;
integer m;
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.

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)
cond																	E							

Half-precision scalar (size == 01)
(FEAT_FP16)

```
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;
integer esize;
integer d;
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

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	1	D	1	1	0	1	0	0	Vd	1	0	size	1	1	M	0	Vm
																				E				

Half-precision scalar (size == 01)
(FEAT_FP16)

```
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;
integer esize;
integer d;
integer m;
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.

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	0	1	1	1	0	1	D	1	1	0	1	0	1	Vd				1	0	size		1	1	(0)	0	(0)	(0)	(0)	(0)
																								E							

Half-precision scalar (size == 01)
(FEAT_FP16)

```
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' && InITBlock()  then UNPREDICTABLE;
quiet_nan_exc = (E == '1');  with_zero = TRUE;
integer esize;
integer d;
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' && 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', 16) 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', 32) else S[m];
            nzcv = FPCompare(S[d], op32, quiet_nan_exc, FPSCR[]);
        when 64
            bits(64) op64 = if with_zero then FPZero('0', 64) else D[m];
            nzcv = FPCompare(D[d], op64, quiet_nan_exc, FPSCR[]);

    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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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	D	1	1	size		0	0	Vd				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	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	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[D[d+r],e,esize] = BitCount(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_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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

31 30 29 28	27	26	25	24	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	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>
```

```
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

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	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>
```

```
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.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckVFPEnabled(TRUE);
    if double_to_single then
        S[d] = FPConvert(D[m], FPSCR[], 32);
    else
        D[d] = FPConvert(S[m], FPSCR[], 64);
```

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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

31	30	29	28	27	26	25	24	23	22	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	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 IN {'000xxx'} then SEE "Related encodings";
if op<1> == '0' && !HaveFP16Ext() then UNDEFINED;
if op<1> == '0' && imm6 IN {'10xxxx'} then UNDEFINED;
if imm6 IN {'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');
integer esize;
integer elements;
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

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	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 IN {'000xxx'} then SEE "Related encodings";
if op<1> == '0' && !HaveFP16Ext() then UNDEFINED;
if op<1> == '0' && imm6 IN {'10xxxx'} then UNDEFINED;
if imm6 IN {'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');
integer esize;
integer elements;
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;
```

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*.

<dt1> Is the data type for the elements of the destination vector, encoded in “op:U”:

op	U	<dt1>
00	x	F16
01	0	S16
01	1	U16
10	x	F32
11	0	S32
11	1	U32

<dt2> Is the data type for the elements of the source vector, encoded in “op:U”:

op	U	<dt2>
00	0	S16
00	1	U16
01	x	F16
10	0	S32
10	1	U32
11	x	F32

<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)*.

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_fixed then
                result = FPToFixed(op1, frac_bits, unsigned, StandardFPSCRValue(),
                                   FPRounding_ZERO, esize);
            else
                result = FixedToFP(op1, frac_bits, unsigned, StandardFPSCRValue(),
                                   FPRounding_TIEEVEN, esize);
            Elem[D[d+r],e,esize] = result;
```

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)
(FEAT_FP16)

```
VCVT{<c>}{<q>}.F16.<dt> <Sdm>, <Sdm>, #<fbits>
```

Half-precision scalar (op == 1 && sf == 01)
(FEAT_FP16)

```
VCVT{<c>}{<q>}.<dt>.F16 <Sdm>, <Sdm>, #<fbits>
```

Single-precision scalar (op == 0 && sf == 10)

```
VCVT{<c>}{<q>}.F32.<dt> <Sdm>, <Sdm>, #<fbits>
```

Single-precision scalar (op == 1 && sf == 10)

```
VCVT{<c>}{<q>}.<dt>.F32 <Sdm>, <Sdm>, #<fbits>
```

Double-precision scalar (op == 0 && sf == 11)

```
VCVT{<c>}{<q>}.F64.<dt> <Ddm>, <Ddm>, #<fbits>
```

Double-precision scalar (op == 1 && sf == 11)

```
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);
integer fp_size;
integer d;
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;
```

CONSTRAINED UNPREDICTABLE behavior

If `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

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	1	D	1	1	1	op	1	U	Vd	1	0	sf	sx	1	i	0	imm4

Half-precision scalar (op == 0 && sf == 01)
(FEAT_FP16)

```
VCVT{<c>}{<q>}.F16.<dt> <Sdm>, <Sdm>, #<fbits>
```

Half-precision scalar (op == 1 && sf == 01)
(FEAT_FP16)

```
VCVT{<c>}{<q>}.<dt>.F16 <Sdm>, <Sdm>, #<fbits>
```

Single-precision scalar (op == 0 && sf == 10)

```
VCVT{<c>}{<q>}.F32.<dt> <Sdm>, <Sdm>, #<fbits>
```

Single-precision scalar (op == 1 && sf == 10)

```
VCVT{<c>}{<q>}.<dt>.F32 <Sdm>, <Sdm>, #<fbits>
```

Double-precision scalar (op == 0 && sf == 11)

```
VCVT{<c>}{<q>}.F64.<dt> <Ddm>, <Ddm>, #<fbits>
```

Double-precision scalar (op == 1 && sf == 11)

```
VCVT{<c>}{<q>}.<dt>.F64 <Ddm>, <Ddm>, #<fbits>
```

```
if sf == '00' || (sf == '01' && !HaveFP16Ext()) then UNDEFINED;
if sf == '01' && InITBlock() then UNPREDICTABLE;
to_fixed = (op == '1');  unsigned = (U == '1');
size = if sx == '0' then 16 else 32;
frac_bits = size - UInt(imm4:i);
integer fp_size;
integer d;
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;
```

CONSTRAINED UNPREDICTABLE behavior

If `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 CONSTRAINED 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”:

U	sx	<dt>
0	0	S16
0	1	S32
1	0	U16
1	1	U32

<Sdm> Is the 32-bit name of the SIMD&FP destination and source register, encoded in the "Vd:D" field.

<Ddm> Is the 64-bit name of the SIMD&FP destination and source register, encoded in the "D:Vd" 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);
    if to_fixed then
        bits(size) result;
        case fp_size of
            when 16
                result = FPToFixed(S[d]<15:0>, frac_bits, unsigned, FPSCR[],
                                   FPRounding_ZERO, size);
                S[d] = Extend(result, 32, unsigned);
            when 32
                result = FPToFixed(S[d], frac_bits, unsigned, FPSCR[], FPRounding_ZERO, size);
                S[d] = Extend(result, 32, unsigned);
            when 64
                result = FPToFixed(D[d], frac_bits, unsigned, FPSCR[], FPRounding_ZERO, size);
                D[d] = Extend(result, 64, unsigned);
    else
        case fp_size of
            when 16
                bits(16) fp16 = FixedToFP(S[d]<size-1:0>, frac_bits, unsigned, FPSCR[],
                                          FPRounding_TIEEVEN, 16);
                S[d] = Zeros(16):fp16;
            when 32
                S[d] = FixedToFP(S[d]<size-1:0>, frac_bits, unsigned, FPSCR[],
                                 FPRounding_TIEEVEN, 32);
            when 64
                D[d] = FixedToFP(D[d]<size-1:0>, frac_bits, unsigned, FPSCR[],
                                 FPRounding_TIEEVEN, 64);
```

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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');
integer esize;
integer elements;
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	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	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');
integer esize;
integer elements;
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”:

size	op	<dt1>
01	0x	F16
01	10	S16
01	11	U16
10	0x	F32
10	10	S32
10	11	U32

<dt2> Is the data type for the elements of the source vector, encoded in “size:op”:

size	op	<dt2>
01	00	S16
01	01	U16
01	1x	F16
10	00	S32
10	01	U32
10	1x	F32

<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, esize);
            else
                result = FixedToFP(op1, 0, unsigned, StandardFPSCRValue(),
                                   FPRounding_TIEEVEN, esize);
            Elem[D[d+r],e,esize] = result;
```

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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	0	0	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;
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	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;
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.

<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.

<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
        if half_to_single then
            Elem[Q[d>>1],e,32] = FPConvert(Elem[Din[m],e,16], StandardFPSCRValue(), 32);
        else
            Elem[D[d],e,16] = FPConvert(Elem[Qin[m>>1],e,32], StandardFPSCRValue(), 16);
```

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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

31 30 29 28	27	26	25	24	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	1	0	x	Vd	1	0	size	1	1	M	0	Vm
cond										opc2							op				

Half-precision scalar (opc2 == 100 && size == 01)
(FEAT_FP16)

```
VCVT{<c>}{<q>}.U32.F16 <Sd>, <Sm>
```

Half-precision scalar (opc2 == 101 && size == 01)
(FEAT_FP16)

```
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 IN {'10x'}) then SEE "Related encodings";
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && cond != '1110' then UNPREDICTABLE;
integer d;
integer esize;
integer m;
boolean unsigned;
FPRounding rounding;
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

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	1	D	1	1	1	1	0	x	Vd				1	0	size		1	1	M	0	Vm			
													opc2											op							

Half-precision scalar (opc2 == 100 && size == 01)
(FEAT_FP16)

```
VCVT{<c>}{<q>}.U32.F16 <Sd>, <Sm>
```

Half-precision scalar (opc2 == 101 && size == 01)
(FEAT_FP16)

```
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 IN {'10x'}) then SEE "Related encodings";
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && InITBlock()  then UNPREDICTABLE;
integer esize;
integer m;
integer d;
boolean unsigned;
FPRounding rounding;
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' && 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

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckVFPEnabled(TRUE);
    if to_integer then
        case esize of
            when 16
                S[d] = FPToFixed(S[m]<15:0>, 0, unsigned, FPSCR[], rounding, 32);
            when 32
                S[d] = FPToFixed(S[m], 0, unsigned, FPSCR[], rounding, 32);
            when 64
                S[d] = FPToFixed(D[m], 0, unsigned, FPSCR[], rounding, 32);
    else
        case esize of
            when 16
                bits(16) fp16 = FixedToFP(S[m], 0, unsigned, FPSCR[], rounding, 16);
                S[d] = Zeros(16):fp16;
            when 32
                S[d] = FixedToFP(S[m], 0, unsigned, FPSCR[], rounding, 32);
            when 64
                D[d] = FixedToFP(S[m], 0, unsigned, FPSCR[], rounding, 64);
```

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
(FEAT_AA32BF16)

31	30	29	28	27	26	25	24	23	22	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	1	1	0	Vd	0	1	1	0	0	1	M	0	Vm

A1

```
VCVT{<c>}{<q>}.BF16.F32 <Dd>, <Qm>
```

```
if !HaveAArch32BF16Ext() then UNDEFINED;
if Vm<0> == '1' then UNDEFINED;
integer d = UInt(D:Vd);
integer m = UInt(M:Vm);
```

T1
(FEAT_AA32BF16)

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	1	1	0	Vd	0	1	1	0	0	1	M	0	Vm

T1

```
VCVT{<c>}{<q>}.BF16.F32 <Dd>, <Qm>
```

```
if !HaveAArch32BF16Ext() then UNDEFINED;
if Vm<0> == '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;
```

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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	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)
(FEAT_FP16)

```
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 IN {'10x'}) then SEE "Related encodings";
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && cond != '1110' then UNPREDICTABLE;
integer d;
integer esize;
integer m;
boolean unsigned;
FPRounding rounding;
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

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	1	D	1	1	1	0	0	0	Vd				1	0	size		op	1	M	0	Vm			
													opc2																		

Half-precision scalar (size == 01)
(FEAT_FP16)

```
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 IN {'10x'}) then SEE "Related encodings";
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && InITBlock()  then UNPREDICTABLE;
integer esize;
integer m;
integer d;
boolean unsigned;
FPRounding rounding;
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' && 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”:

op	<dt>
0	U32
1	S32

<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.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckVFPEnabled(TRUE);
    if to_integer then
        case esize of
            when 16
                S[d] = FPToFixed(S[m]<15:0>, 0, unsigned, FPSCR[], rounding, 32);
            when 32
                S[d] = FPToFixed(S[m], 0, unsigned, FPSCR[], rounding, 32);
            when 64
                S[d] = FPToFixed(D[m], 0, unsigned, FPSCR[], rounding, 32);
    else
        case esize of
            when 16
                bits(16) fp16 = FixedToFP(S[m], 0, unsigned, FPSCR[], rounding, 16);
                S[d] = Zeros(16):fp16;
            when 32
                S[d] = FixedToFP(S[m], 0, unsigned, FPSCR[], rounding, 32);
            when 64
                D[d] = FixedToFP(S[m], 0, unsigned, FPSCR[], rounding, 64);
```

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

31	30	29	28	27	26	25	24	23	22	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	0	0	0	op	Q	M	0	Vm			
																						RM									

64-bit SIMD vector (Q == 0)

```
VCVTA{<q>}.<dt>.<dt2> <Dd>, <Dm>
```

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');
integer esize;
integer elements;
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	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	1	Vd				0	0	0	0	op	Q	M	0	Vm			
																						RM									

64-bit SIMD vector (Q == 0)

```
VCVTA{<q>}.<dt>.<dt2> <Dd>, <Dm>
```

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');
integer esize;
integer elements;
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”:

op	size	<dt>
0	01	S16
0	10	S32
1	01	U16
1	10	U32

<dt2> Is the data type for the elements of the source vector, encoded in “size”:

size	<dt2>
01	F16
10	F32

<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, esize);
```

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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

31	30	29	28	27	26	25	24	23	22	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)
(FEAT_FP16)

```
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 = FPDecodeRM(RM);  unsigned = (op == '0');
d = UInt(Vd:D);
integer esize;
integer m;
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	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)
(FEAT_FP16)

```
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);
integer esize;
integer m;
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”:

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, 32);
    when 32
        S[d] = FPToFixed(S[m], 0, unsigned, FPSCR[], rounding, 32);
    when 64
        S[d] = FPToFixed(D[m], 0, unsigned, FPSCR[], rounding, 32);
```

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>
```

```
uses_double = (sz == '1'); convert_from_half = (op == '0');
lowbit = (if T == '1' then 16 else 0);
integer d;
integer m;
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	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	0	1	op	Vd	1	0	1	sz	0	1	M	0	Vm
																					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>
```

```
uses_double = (sz == '1'); convert_from_half = (op == '0');
lowbit = (if T == '1' then 16 else 0);
integer d;
integer m;
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);
```

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

```
if ConditionPassed() then
    EncodingSpecificOperations(); CheckVFPEnabled(TRUE);
    bits(16) hp;
    if convert_from_half then
        hp = S[m]<lowbit+15:lowbit>;
        if uses_double then
            D[d] = FPConvert(hp, FPSCR[], 64);
        else
            S[d] = FPConvert(hp, FPSCR[], 32);
    else
        if uses_double then
            hp = FPConvert(D[m], FPSCR[], 16);
        else
            hp = FPConvert(S[m], FPSCR[], 16);
        S[d]<lowbit+15:lowbit> = hp;
```

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
(FEAT_AA32BF16)

31	30	29	28	27	26	25	24	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	1	Vd				1	0	0	1	0	1	M	0	Vm			
cond																															

A1

```
VCVTB{<c>}{<q>}.BF16.F32 <Sd>, <Sm>
```

```
if !HaveAArch32BF16Ext() then UNDEFINED;
integer d = UInt(Vd:D);
integer m = UInt(Vm:M);
```

T1
(FEAT_AA32BF16)

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	1	D	1	1	0	0	1	1	Vd				1	0	0	1	0	1	M	0	Vm			

T1

```
VCVTB{<c>}{<q>}.BF16.F32 <Sd>, <Sm>
```

```
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]<15:0> = FPConvertBF(S[m], FPSCR[]);
```

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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
1	1	1	1	0	0	1	1	1	D	1	1	size	1	1	Vd	0	0	1	1	op	Q	M	0	Vm
																		RM						

64-bit SIMD vector (Q == 0)

```
VCVTM{<q>}.<dt>.<dt2> <Dd>, <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 = FPDecodeRM(RM);  unsigned = (op == '1');
integer esize;
integer elements;
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	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	1	Vd	0	0	1	1	op	Q	M	0	Vm
																		RM						

64-bit SIMD vector (Q == 0)

```
VCVTM{<q>}.<dt>.<dt2> <Dd>, <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 = FPDecodeRM(RM);  unsigned = (op == '1');
integer esize;
integer elements;
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”:

op	size	<dt>
0	01	S16
0	10	S32
1	01	U16
1	10	U32

<dt2> Is the data type for the elements of the source vector, encoded in “size”:

size	<dt2>
01	F16
10	F32

<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, esize);
```

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*, *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	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)
(FEAT_FP16)

```
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>, <Dm>
```

```
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
rounding = FPDecodeRM(RM);  unsigned = (op == '0');
d = UInt(Vd:D);
integer esize;
integer m;
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	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)
(FEAT_FP16)

```
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>, <Dm>
```

```
if InITBlock() then UNPREDICTABLE;
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
rounding = FPDecodeRM(RM);  unsigned = (op == '0');
d = UInt(Vd:D);
integer esize;
integer m;
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”:

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, 32);
    when 32
        S[d] = FPToFixed(S[m], 0, unsigned, FPSCR[], rounding, 32);
    when 64
        S[d] = FPToFixed(D[m], 0, unsigned, FPSCR[], rounding, 32);
```

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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	1	0	0	1	1	1	D	1	1	size		1	1	Vd				0	0	0	1	op	Q	M	0	Vm			
																						RM									

64-bit SIMD vector (Q == 0)

```
VCVTN{<q>}.<dt>.<dt2> <Dd>, <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 = FPDecodeRM(RM);  unsigned = (op == '1');
integer esize;
integer elements;
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	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	1	Vd				0	0	0	1	op	Q	M	0	Vm			
																						RM									

64-bit SIMD vector (Q == 0)

```
VCVTN{<q>}.<dt>.<dt2> <Dd>, <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 = FPDecodeRM(RM);  unsigned = (op == '1');
integer esize;
integer elements;
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”:

op	size	<dt>
0	01	S16
0	10	S32
1	01	U16
1	10	U32

<dt2> Is the data type for the elements of the source vector, encoded in “size”:

size	<dt2>
01	F16
10	F32

<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, esize);
```

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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

31	30	29	28	27	26	25	24	23	22	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	1	Vd				1	0	!= 00		op	1	M	0	Vm			
														RM								size									

Half-precision scalar (size == 01)
(FEAT_FP16)

```
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 size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
rounding = FPDecodeRM(RM);  unsigned = (op == '0');
d = UInt(Vd:D);
integer esize;
integer m;
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	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	1	Vd				1	0	!= 00		op	1	M	0	Vm			
														RM								size									

Half-precision scalar (size == 01)
(FEAT_FP16)

```
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 InITBlock() then UNPREDICTABLE;
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
rounding = FPDecodeRM(RM);  unsigned = (op == '0');
d = UInt(Vd:D);
integer esize;
integer m;
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”:

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, 32);
    when 32
        S[d] = FPToFixed(S[m], 0, unsigned, FPSCR[], rounding, 32);
    when 64
        S[d] = FPToFixed(D[m], 0, unsigned, FPSCR[], rounding, 32);
```

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	0	1	1	1	D	1	1	size		1	1	Vd				0	0	1	0	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');
integer esize;
integer elements;
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	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	1	Vd				0	0	1	0	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');
integer esize;
integer elements;
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”:

op	size	<dt>
0	01	S16
0	10	S32
1	01	U16
1	10	U32

<dt2> Is the data type for the elements of the source vector, encoded in “size”:

size	<dt2>
01	F16
10	F32

<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, esize);
```

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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	0	1	D	1	1	1	1	1	0	Vd				1	0	!= 00		op	1	M	0	Vm			
														RM								size									

Half-precision scalar (size == 01)
(FEAT_FP16)

```
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>, <Dm>
```

```
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
rounding = FPDecodeRM(RM);  unsigned = (op == '0');
d = UInt(Vd:D);
integer esize;
integer m;
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	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	0	Vd				1	0	!= 00		op	1	M	0	Vm			
														RM								size									

Half-precision scalar (size == 01)
(FEAT_FP16)

```
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>, <Dm>
```

```
if InITBlock() then UNPREDICTABLE;
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
rounding = FPDecodeRM(RM);  unsigned = (op == '0');
d = UInt(Vd:D);
integer esize;
integer m;
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”:

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, 32);
    when 32
        S[d] = FPToFixed(S[m], 0, unsigned, FPSCR[], rounding, 32);
    when 64
        S[d] = FPToFixed(D[m], 0, unsigned, FPSCR[], rounding, 32);
```

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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	1	0	x	Vd	1	0	size	0	1	M	0	Vm
cond										opc2							op				

Half-precision scalar (opc2 == 100 && size == 01)
(FEAT_FP16)

```
VCVTR{<c>}{<q>}.U32.F16 <Sd>, <Sm>
```

Half-precision scalar (opc2 == 101 && size == 01)
(FEAT_FP16)

```
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>, <Dm>
```

Double-precision scalar (opc2 == 101 && size == 11)

```
VCVTR{<c>}{<q>}.S32.F64 <Sd>, <Dm>
```

```
if opc2 != '000' && !(opc2 IN {'10x'}) then SEE "Related encodings";
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && cond != '1110' then UNPREDICTABLE;
integer d;
integer esize;
integer m;
boolean unsigned;
FPRounding rounding;
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

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	1	D	1	1	1	1	0	x	Vd				1	0	size		0	1	M	0	Vm			
														opc2										op							

Half-precision scalar (opc2 == 100 && size == 01)
(FEAT_FP16)

```
VCVTR{<c>}{<q>}.U32.F16 <Sd>, <Sm>
```

Half-precision scalar (opc2 == 101 && size == 01)
(FEAT_FP16)

```
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>, <Dm>
```

Double-precision scalar (opc2 == 101 && size == 11)

```
VCVTR{<c>}{<q>}.S32.F64 <Sd>, <Dm>
```

```
if opc2 != '000' && !(opc2 IN {'10x'}) then SEE "Related encodings";
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && InITBlock()  then UNPREDICTABLE;
integer esize;
integer m;
integer d;
boolean unsigned;
FPRounding rounding;
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' && 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

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckVFPEnabled(TRUE);
    if to_integer then
        case esize of
            when 16
                S[d] = FPToFixed(S[m]<15:0>, 0, unsigned, FPSCR[], rounding, 32);
            when 32
                S[d] = FPToFixed(S[m], 0, unsigned, FPSCR[], rounding, 32);
            when 64
                S[d] = FPToFixed(D[m], 0, unsigned, FPSCR[], rounding, 32);
    else
        case esize of
            when 16
                bits(16) fp16 = FixedToFP(S[m], 0, unsigned, FPSCR[], rounding, 16);
                S[d] = Zeros(16):fp16;
            when 32
                S[d] = FixedToFP(S[m], 0, unsigned, FPSCR[], rounding, 32);
            when 64
                D[d] = FixedToFP(S[m], 0, unsigned, FPSCR[], rounding, 64);
```

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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

31-28	27	26	25	24	23	22	21	20	19	18	17	16	15-12	11	10	9	8	7	6	5	4	3-0
!= 1111	1	1	1	0	1	D	1	1	0	0	1	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);
integer d;
integer m;
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	15-12	11	10	9	8	7	6	5	4	3-0
1	1	1	0	1	1	1	0	1	D	1	1	0	0	1	op	Vd	1	0	1	sz	1	1	M	0	Vm
																					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);
integer d;
integer m;
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);
```

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

```
if ConditionPassed() then
    EncodingSpecificOperations(); CheckVFPEnabled(TRUE);
    bits(16) hp;
    if convert_from_half then
        hp = S[m]<lowbit+15:lowbit>;
        if uses_double then
            D[d] = FPConvert(hp, FPSCR[], 64);
        else
            S[d] = FPConvert(hp, FPSCR[], 32);
    else
        if uses_double then
            hp = FPConvert(D[m], FPSCR[], 16);
        else
            hp = FPConvert(S[m], FPSCR[], 16);
        S[d]<lowbit+15:lowbit> = hp;
```

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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

(FEAT_AA32BF16)

31 30 29 28	27	26	25	24	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	1	Vd	1	0	0	1	1	1	M	0	Vm
cond																						

A1

```
VCVTT{<c>}{<q>}.BF16.F32 <Sd>, <Sm>
```

```
if !HaveAArch32BF16Ext() then UNDEFINED;
integer d = UInt(Vd:D);
integer m = UInt(Vm:M);
```

T1

(FEAT_AA32BF16)

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	1	D	1	1	0	0	1	1	Vd	1	0	0	1	1	1	M	0	Vm

T1

```
VCVTT{<c>}{<q>}.BF16.F32 <Sd>, <Sm>
```

```
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> = FPConvertBF(S[m], FPSCR[]);
```

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

VDIV

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)
(FEAT_FP16)

```
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;
integer esize;
integer d;
integer n;
integer m;
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	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)
(FEAT_FP16)

```
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;
integer esize;
integer d;
integer n;
integer m;
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
            S[d] = Zeros(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 (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

(FEAT_AA32BF16)

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

(FEAT_AA32BF16)

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

<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;
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 = FPAdd_BF16(BFMulH(elt1_a, elt2_a), BFMulH(elt1_b, elt2_b));
        Elem[result, e, 32] = FPAdd_BF16(Elem[result, e, 32], sum);
    D[d+r] = result;
```

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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

(FEAT_AA32BF16)

31	30	29	28	27	26	25	24	23	22	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	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

(FEAT_AA32BF16)

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

```
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 = FPAdd_BF16(BFMulH(elt1_a, elt2_a), BFMulH(elt1_b, elt2_b));
        Elem[result, e, 32] = FPAdd_BF16(Elem[result, e, 32], sum);
    D[d+r] = result;
```

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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)
cond																						

A1

```
VDUP{<c>}{<q>}.<size> <Qd>, <Rt> // (Encoded as Q = 1)

VDUP{<c>}{<q>}.<size> <Dd>, <Rt> // (Encoded as Q = 0)
```

```
if Q == '1' && Vd<0> == '1' then UNDEFINED;
d = UInt(D:Vd);  t = UInt(Rt);  regs = if Q == '0' then 1 else 2;
integer esize;
integer elements;
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	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	B	Q	0	Vd	Rt	1	0	1	1	D	0	E	1	(0)	(0)	(0)	(0)

T1

```
VDUP{<c>}{<q>}.<size> <Qd>, <Rt> // (Encoded as Q = 1)

VDUP{<c>}{<q>}.<size> <Dd>, <Rt> // (Encoded as Q = 0)
```

```
if Q == '1' && Vd<0> == '1' then UNDEFINED;
d = UInt(D:Vd);  t = UInt(Rt);  regs = if Q == '0' then 1 else 2;
integer esize;
integer elements;
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*.

<size> 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.

<Qd> The destination vector for a quadword operation.

<Dd> The destination vector for a doubleword operation.

<Rt> The Arm source register.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    scalar = R[t]<esize-1:0>;
    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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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
1	1	1	1	0	0	1	1	1	D	1	1	imm4				Vd				1	1	0	0	0	Q	M	0	Vm			

(Q == 0)

```
VDUP{<c>}{<q>}.<size> <Dd>, <Dm[x]>
```

(Q == 1)

```
VDUP{<c>}{<q>}.<size> <Qd>, <Dm[x]>
```

```
if imm4 IN {'x000'} then UNDEFINED;
if Q == '1' && Vd<0> == '1' then UNDEFINED;
integer esize;
integer elements;
integer index;
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	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	imm4				Vd				1	1	0	0	0	Q	M	0	Vm			

(Q == 0)

```
VDUP{<c>}{<q>}.<size> <Dd>, <Dm[x]>
```

(Q == 1)

```
VDUP{<c>}{<q>}.<size> <Qd>, <Dm[x]>
```

```
if imm4 IN {'x000'} then UNDEFINED;
if Q == '1' && Vd<0> == '1' then UNDEFINED;
integer esize;
integer elements;
integer index;
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

```
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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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

31	30	29	28	27	26	25	24	23	22	21	20	19-16	15-12	11	10	9	8	7	6	5	4	3-0
1	1	1	1	0	0	1	1	0	D	0	0	Vn	Vd	0	0	0	1	N	Q	M	1	Vm

64-bit SIMD vector (Q == 0)

```
VEOR{<c>}{<q>}{.<dt>} {<Dd>,} <Dn>, <Dm>
```

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

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

```
VEOR{<c>}{<q>}{.<dt>} {<Dd>,} <Dn>, <Dm>
```

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;
```

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] 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.

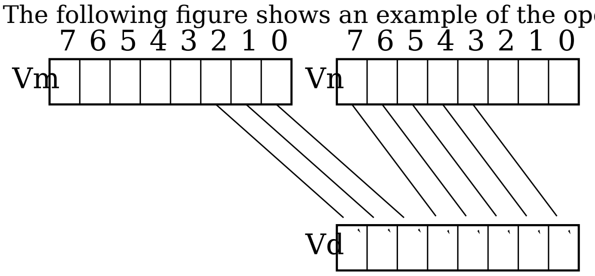
Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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	19-16	15-12	11-8	7	6	5	4	3-0
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>, <Dm>, #<imm>
```

128-bit SIMD vector (Q == 1)

```
VEXT{<c>}{<q>}.8 {<Qd>,} <Qn>, <Qm>, #<imm>
```

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if Q == '0' && imm4<3> == '1' then UNDEFINED;
quadword_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	3-0	15-12	11-8	7	6	5	4	3-0
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>, <Dm>, #<imm>
```

128-bit SIMD vector (Q == 1)

```
VEXT{<c>}{<q>}.8 {<Qd>,} <Qn>, <Qm>, #<imm>
```

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if Q == '0' && imm4<3> == '1' then UNDEFINED;
quadword_operation = (Q == '1');  position = 8 * UInt(imm4);
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*.
<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

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    if quadword_operation then
        Q[d>>1] = (Q[m>>1]:Q[n>>1])<position+127:position>;
    else
        D[d] = (D[m]:D[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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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, any CONSTRAINED UNPREDICTABLE behavior, and any operational information 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	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>}.<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*.

<size>	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 *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

31–25	24	23	22	21	20	19–16	15–12	11–8	7	6	5	4	3–0
1 1 1 1 0 0 1	0	0	D	0	sz	Vn	Vd	1 1 0 0	N	Q	M	1	Vm
				op									

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');
integer esize;
integer elements;
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

31–28	27–24	23	22	21–20	19–16	15–12	11–10	9–8	7	6	5	4	3–0
!= 1111	1 1 1 0	1	D	1 0	Vn	Vd	1 0	size	N	0	M	0	Vm
cond										op			

Half-precision scalar (size == 01)
(FEAT_FP16)

```
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');
integer esize;
integer d;
integer n;
integer m;
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	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	sz	Vn				Vd				1	1	0	0	N	Q	M	1	Vm			
										op																					

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');
integer esize;
integer elements;
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	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	0	Vn	Vd	1	0	size	N	0	M	0	Vm
																		op			

Half-precision scalar (size == 01)
(FEAT_FP16)

```
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');
integer esize;
integer d;
integer n;
integer m;
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”:

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[]);
```

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
(FEAT_AA32BF16)

31	30	29	28	27	26	25	24	23	22	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	1	Vn				Vd				1	0	0	0	N	Q	M	1	Vm			

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>);
integer i = UInt(M:Vm<3>);
integer elements = 128 DIV 32;
integer sel = UInt(Q);
```

T1
(FEAT_AA32BF16)

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	0	D	1	1	Vn				Vd				1	0	0	0	N	Q	M	1	Vm			

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>);
integer i = UInt(M:Vm<3>);
integer elements = 128 DIV 32;
integer sel = UInt(Q);
```

Assembler Symbols

<bt> Is the bottom or top element specifier, encoded in “Q”:

Q	<bt>
0	B
1	T

<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.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "Vm<2:0>" field.

<index> Is the element index in the range 0 to 3, encoded in the "M:Vm<3>" field.

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;
```

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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

(FEAT_AA32BF16)

31	30	29	28	27	26	25	24	23	22	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				Vd				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

(FEAT_AA32BF16)

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				Vd				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”:

Q	<bt>
0	B
1	T

<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;
```

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.

Note

ID_ISAR6.FHM indicates whether this instruction is supported.

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

A1
(FEAT_FHM)

31	30	29	28	27	26	25	24	23	22	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	0	0	Vn	Vd	1	0	0	0	N	Q	M	1	Vm
											S											

64-bit SIMD vector (Q == 0)

```
VFMAL{<q>}.F16 <Dd>, <Sn>, <Sm>[<index>]
```

128-bit SIMD vector (Q == 1)

```
VFMAL{<q>}.F16 <Qd>, <Dn>, <Dm>[<index>]
```

```
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';
```

T1
(FEAT_FHM)

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	0	D	0	0	Vn	Vd	1	0	0	0	N	Q	M	1	Vm
											S											

64-bit SIMD vector (Q == 0)

```
VFMAL{<q>}.F16 <Dd>, <Sn>, <Sm>[<index>]
```

128-bit SIMD vector (Q == 1)

```
VFMAL{<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

<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.
<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,
                                           StandardFPSCRValue());
    D[d+r] = 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.

Note

ID_ISAR6.FHM indicates whether this instruction is supported.

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

A1

(FEAT_FHM)

31	30	29	28	27	26	25	24	23	22	21	20	19-16	15-12	11	10	9	8	7	6	5	4	3-0
1	1	1	1	1	1	0	0	0	D	1	0	Vn	Vd	1	0	0	0	N	Q	M	1	Vm
								S														

64-bit SIMD vector (Q == 0)

```
VFMAL{<q>}.F16 <Dd>, <Sn>, <Sm>
```

128-bit SIMD vector (Q == 1)

```
VFMAL{<q>}.F16 <Qd>, <Dn>, <Dm>
```

```
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(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

(FEAT_FHM)

15	14	13	12	11	10	9	8	7	6	5	4	3-0	15-12	11	10	9	8	7	6	5	4	3-0
1	1	1	1	1	1	0	0	0	D	1	0	Vn	Vd	1	0	0	0	N	Q	M	1	Vm
								S														

64-bit SIMD vector (Q == 0)

```
VFMAL{<q>}.F16 <Dd>, <Sn>, <Sm>
```

128-bit SIMD vector (Q == 1)

```
VFMAL{<q>}.F16 <Qd>, <Dn>, <Dm>
```

```
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(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.
<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

```
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>;
for r = 0 to regs-1
    operand3 = D[d+r];
    for e = 0 to 1
        element1 = Elem[operand1, 2*r+e, esize DIV 2];
        element2 = Elem[operand2, 2*r+e, esize DIV 2];
        if sub_op then element1 = FPNeg(element1);
        Elem[result, e, esize] = FPMulAddH(Elem[operand3, e, esize], element1, element2,
                                           StandardFPSCRValue());
    D[d+r] = result;
```

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*, *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	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;
advsimd = TRUE; op1_neg = (op == '1');
integer esize;
integer elements;
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	D	1	0	Vn				Vd				1	0	size		N	1	M	0	Vm			
cond																									op						

Half-precision scalar (size == 01)
(FEAT_FP16)

```
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' && cond != '1110' then UNPREDICTABLE;
advsimd = FALSE; op1_neg = (op == '1');
integer esize;
integer d;
integer n;
integer m;
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	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	0	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');
integer esize;
integer elements;
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	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	0	Vn				Vd				1	0	size		N	1	M	0	Vm			
																									op						

Half-precision scalar (size == 01)
(FEAT_FP16)

```
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');
integer esize;
integer d;
integer n;
integer m;
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”:

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_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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*, *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.

Note

ID_ISAR6.FHM indicates whether this instruction is supported.

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

A1
(FEAT_FHM)

31	30	29	28	27	26	25	24	23	22	21	20	19-16	15-12	11	10	9	8	7	6	5	4	3-0
1	1	1	1	1	1	1	0	0	D	0	1	Vn	Vd	1	0	0	0	N	Q	M	1	Vm
											S											

64-bit SIMD vector (Q == 0)

```
VFMSL{<q>}.F16 <Dd>, <Sn>, <Sm>[<index>]
```

128-bit SIMD vector (Q == 1)

```
VFMSL{<q>}.F16 <Qd>, <Dn>, <Dm>[<index>]
```

```
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';
```

T1
(FEAT_FHM)

15	14	13	12	11	10	9	8	7	6	5	4	3-0	15-12	11	10	9	8	7	6	5	4	3-0
1	1	1	1	1	1	1	0	0	D	0	1	Vn	Vd	1	0	0	0	N	Q	M	1	Vm
											S											

64-bit SIMD vector (Q == 0)

```
VFMSL{<q>}.F16 <Dd>, <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

<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.
<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,
                                           StandardFPSCRValue());
    D[d+r] = result;
```

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.

Note

ID_ISAR6.FHM indicates whether this instruction is supported.

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

A1
(FEAT_FHM)

31	30	29	28	27	26	25	24	23	22	21	20	19-16	15-12	11	10	9	8	7	6	5	4	3-0
1	1	1	1	1	1	0	0	1	D	1	0	Vn	Vd	1	0	0	0	N	Q	M	1	Vm
								S														

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 !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';
```

T1
(FEAT_FHM)

15	14	13	12	11	10	9	8	7	6	5	4	3-0	15-12	11	10	9	8	7	6	5	4	3-0
1	1	1	1	1	1	0	0	1	D	1	0	Vn	Vd	1	0	0	0	N	Q	M	1	Vm
								S														

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 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(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.
<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

```
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>;
for r = 0 to regs-1
    operand3 = D[d+r];
    for e = 0 to 1
        element1 = Elem[operand1, 2*r+e, esize DIV 2];
        element2 = Elem[operand2, 2*r+e, esize DIV 2];
        if sub_op then element1 = FPNeg(element1);
        Elem[result, e, esize] = FPMulAddH(Elem[operand3, e, esize], element1, element2,
                                            StandardFPSCRValue());
    D[d+r] = result;
```

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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 *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	1	M	0	Vm
cond															op			

Half-precision scalar (size == 01)
(FEAT_FP16)

```
VFNMA{<c>}{<q>}.F16 <Sd>, <Sn>, <Sm>
```

Single-precision scalar (size == 10)

```
VFNMA{<c>}{<q>}.F32 <Sd>, <Sn>, <Sm>
```

Double-precision scalar (size == 11)

```
VFNMA{<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');
integer esize;
integer d;
integer n;
integer m;
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	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	1	M	0	Vm
											op			

Half-precision scalar (size == 01)
(FEAT_FP16)

```
VFNMA{<c>}{<q>}.F16 <Sd>, <Sn>, <Sm>
```

Single-precision scalar (size == 10)

```
VFNMA{<c>}{<q>}.F32 <Sd>, <Sn>, <Sm>
```

Double-precision scalar (size == 11)

```
VFNMA{<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');
integer esize;
integer d;
integer n;
integer m;
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[]);
```

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)
(FEAT_FP16)

```
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');
integer esize;
integer d;
integer n;
integer m;
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	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)
(FEAT_FP16)

```
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');
integer esize;
integer d;
integer n;
integer m;
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> {<Dd>, }<Dn>, <Dm>
```

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	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	0	Vm			
																						op									

64-bit SIMD vector (Q == 0)

```
VHADD{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

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*.

<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.

<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 = 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) >> 1;
            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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

VHSUB

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) .

A1

31	30	29	28	27	26	25	24	23	22	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	0	Vm			
																						op									

64-bit SIMD vector (Q == 0)

```
VHSUB{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VHSUB{<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	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	0	Vm			
																						op									

64-bit SIMD vector (Q == 0)

```
VHSUB{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VHSUB{<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*.

<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.

<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 = 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) >> 1;
            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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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 *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
(FEAT_FP16)

31	30	29	28	27	26	25	24	23	22	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	1	1	M	0	Vm			

A1

```
VINS{<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
(FEAT_FP16)

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	1	D	1	1	0	0	0	0	Vd				1	0	1	0	1	1	M	0	Vm			

T1

```
VINS{<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] = S[m]<15:0> : S[d]<15:0>;
```

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
(FEAT_JSCVT)

31-28	27	26	25	24	23	22	21	20	19	18	17	16	15-12	11	10	9	8	7	6	5	4	3-0
!= 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;
d = UInt(Vd:D);  m = UInt(M:Vm);
```

T1
(FEAT_JSCVT)

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

T1

```
VJCVT{<q>}.S32.F64 <Sd>, <Dm>
```

```
if !HaveFJCVTZSExt() then UNDEFINED;
if InITBlock() then UNPREDICTABLE;
d = UInt(Vd:D);  m = 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();
CheckVFPEnabled(TRUE);
bits(64) fltval = D[m];
bits(32) intval;
bit      Z;
(intval, Z) = FPToFixedJS(fltval, FPSCR[], FALSE, 32);
FPSCR<31:28> = '0':Z:'00';
S[d] = intval;
```

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

31	30	29	28	27	26	25	24	23	22	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	1	1	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 = 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 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	0	D	1	0	Rn				Vd				1	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 = 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	D	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

31	30	29	28	27	26	25	24	23	22	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	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);
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.

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	1	1	1	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 = 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 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	0	D	1	0	Rn				Vd				1	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 = 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.

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	0	D	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.

T4

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	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);
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 (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”:

size	<size>
00	8
01	16
10	32
11	64

<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 the variants of this instruction, see *Advanced SIMD addressing mode*.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();

    address = R[n];

    boolean nontemporal = FALSE;
    boolean tagchecked  = FALSE;
    AccessDescriptor accdesc = CreateAccDescASIMD(MemOp_LOAD, nontemporal, tagchecked);
    if !IsAligned(address, alignment) then
        AArch32.Abort(address, AlignmentFault(accdesc));

    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
                if !IsAligned(address, ebytes) && AlignmentEnforced() then
                    AArch32.Abort(address, AlignmentFault(accdesc));

                if BigEndian(AccessType_ASIMD) then
                    data<31:0> = MemU[address+4,4];
                    data<63:32> = MemU[address,4];
                else
                    data<31:0> = MemU[address,4];
                    data<63:32> = MemU[address+4,4];

            Elem[D[d+r],e,8*ebytes] = data;
            address = address + ebytes;
    if wback then
        if register_index then
            R[n] = R[n] + R[m];
        else
            R[n] = R[n] + 8*regs;
```

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

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

31	30	29	28	27	26	25	24	23	22	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	0	size		T	a	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' || (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.

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	D	1	0	Rn				Vd				1	1	0	0	size		T	a	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' || (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”:

size	<size>
00	8
01	16
10	32
11	RESERVED

<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.

: 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];

    boolean nontemporal = FALSE;
    boolean tagchecked  = FALSE;
    AccessDescriptor accdesc = CreateAccDescASIMD(MemOp_LOAD, nontemporal, tagchecked);
    if !IsAligned(address, alignment) then
        AArch32.Abort(address, AlignmentFault(accdesc));

    constant integer esize = 8 * ebytes;
    bits(esize) element = MemU[address,ebytes];
    bits(64) replicated_element = Replicate(element, 64 DIV esize);
    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;
```

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_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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	19-16	15-12	11	10	9	8	7-4	3-0
1	1	1	1	0	1	0	0	1	D	1	0	Rn	Vd	0	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<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	19-16	15-12	11	10	9	8	7-4	3-0
1	1	1	1	0	1	0	0	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;
```

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	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;
```

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	D	1	0	Rn				Vd				0	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<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”:

size	<size>
00	8
01	16
10	32

<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 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];

    boolean nontemporal = FALSE;
    boolean tagchecked  = FALSE;
    AccessDescriptor accdesc = CreateAccDescASIMD(MemOp_LOAD, nontemporal, tagchecked);
    if !IsAligned(address, alignment) then
        AArch32.Abort(address, AlignmentFault(accdesc));

    Elem[D[d],index,8*ebytes] = MemU[address,ebytes];
    if wback then
        if register_index then
            R[n] = R[n] + R[m];
        else
            R[n] = R[n] + ebytes;
```

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_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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	19-16	15-12	11	10	9	8	7-6	5-4	3-0
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>
```

```
pairs = 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+pairs > 32 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `d2+pairs > 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	19-16	15-12	11	10	9	8	7-6	5-4	3-0
1	1	1	1	0	1	0	0	0	D	1	0	Rn	Vd	0	0	1	1	size	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>
```

```
pairs = 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+pairs > 32 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `d2+pairs > 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

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				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>
```

```
pairs = 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+pairs > 32 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `d2+pairs > 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

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	1	1	size		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>
```

```
pairs = 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+pairs > 32 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `d2+pairs > 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”:

size	<size>
00	8
01	16
10	32
11	RESERVED

<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];

    boolean nontemporal = FALSE;
    boolean tagchecked  = FALSE;
    AccessDescriptor accdesc = CreateAccDescASIMD(MemOp_LOAD, nontemporal, tagchecked);
    if !IsAligned(address, alignment) then
        AArch32.Abort(address, AlignmentFault(accdesc));

    for r = 0 to pairs-1
        for e = 0 to elements-1
            Elem[D[d+r], e,8*ebytes] = MemU[address,ebytes];
            Elem[D[d2+r],e,8*ebytes] = 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*pairs;
```

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_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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	19-16	15-12	11	10	9	8	7-6	5	4	3-0
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	3-0	15-12	11	10	9	8	7-6	5	4	3-0
1	1	1	1	1	0	0	1	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.

For more information about the CONSTRAINED 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”:

size	<size>
00	8
01	16
10	32
11	RESERVED

<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.

<size> == 16
<align> is 32, meaning 32-bit alignment, encoded in the "a" field as 1.

<size> == 32
<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

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();

    address = R[n];

    boolean nontemporal = FALSE;
    boolean tagchecked  = FALSE;
    AccessDescriptor accdesc = CreateAccDescASIMD(MemOp_LOAD, nontemporal, tagchecked);
    if !IsAligned(address, alignment) then
        AArch32.Abort(address, AlignmentFault(accdesc));

    constant integer esize = 8 * ebytes;
    bits(esize) element1 = MemU[address, ebytes];
    bits(esize) element2 = MemU[address+ebytes, ebytes];
    D[d] = Replicate(element1, 64 DIV esize);
    D[d2] = Replicate(element2, 64 DIV esize);

    if wback then
        if register_index then
            R[n] = R[n] + R[m];
        else
            R[n] = R[n] + 2*ebytes;
```

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_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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	19-16	15-12	11	10	9	8	7-4	3-0
1	1	1	1	0	1	0	0	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.

A2

31	30	29	28	27	26	25	24	23	22	21	20	19-16	15-12	11	10	9	8	7-4	3-0
1	1	1	1	0	1	0	0	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;
```

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.

A3

31	30	29	28	27	26	25	24	23	22	21	20	19-16	15-12	11	10	9	8	7-4	3-0
1	1	1	1	0	1	0	0	1	D	1	0	Rn	Vd	1	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)";
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`.

- 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	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	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 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;
```

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.

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	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)";
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`.
- 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 (single 2-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 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];

    boolean nontemporal = FALSE;
    boolean tagchecked  = FALSE;
    AccessDescriptor accdesc = CreateAccDescASIMD(MemOp_LOAD, nontemporal, tagchecked);
    if !IsAligned(address, alignment) then
        AArch32.Abort(address, AlignmentFault(accdesc));

    Elem[D[d], index,8*ebytes] = MemU[address,ebytes];
    Elem[D[d2],index,8*ebytes] = MemU[address+ebytes,ebytes];
    if wback then
        if register_index then
            R[n] = R[n] + R[m];
        else
            R[n] = R[n] + 2*ebytes;
```

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_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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			
																				itype											

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>
```

```
integer inc;
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 << 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`.
- 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	1	0	x	size		align		Rm			
																				itype											

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>
```

```
integer inc;
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 << 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`.
- 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*.

<size> Is the data size, encoded in “size”:

size	<size>
00	8
01	16
10	32
11	RESERVED

<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> }
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.

<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];

    boolean nontemporal = FALSE;
    boolean tagchecked  = FALSE;
    AccessDescriptor accdesc = CreateAccDescASIMD(MemOp_LOAD, nontemporal, tagchecked);
    if !IsAligned(address, alignment) then
        AArch32.Abort(address, AlignmentFault(accdesc));

    for e = 0 to elements-1
        Elem[D[d], e,8*ebytes] = MemU[address,ebytes];
        Elem[D[d2],e,8*ebytes] = MemU[address+ebytes,ebytes];
        Elem[D[d3],e,8*ebytes] = 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;
```

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_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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	1	0	1	0	0	1	D	1	0	Rn	Vd	1	1	1	0	size	T	0	Rm
																				a	

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.

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	D	1	0	Rn	Vd	1	1	1	0	size	T	0	Rm
																				a	

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”:

size	<size>
00	8
01	16
10	32
11	RESERVED

<list> Is a list containing the 64-bit names of three SIMD&FP registers.

The list must be one of:

{ <Dd>[], <Dd+1>[], <Dd+2>[] }
Single-spaced registers, encoded in the "T" field as 0.

{ <Dd>[], <Dd+2>[], <Dd+4>[] }
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*.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    address = R[n];
    constant integer esize = ebytes * 8;
    bits(esize) element1 = MemU[address, ebytes];
    bits(esize) element2 = MemU[address+ebytes,ebytes];
    bits(esize) element3 = MemU[address+2*ebytes,ebytes];

    D[d] = Replicate(element1, 64 DIV esize);
    D[d2] = Replicate(element2, 64 DIV esize);
    D[d3] = Replicate(element3, 64 DIV esize);
    if wback then
        if register_index then
            R[n] = R[n] + R[m];
        else
            R[n] = R[n] + 3*ebytes;
```

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_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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	19-16	15-12	11	10	9	8	7-4	3-0
1	1	1	1	0	1	0	0	1	D	1	0	Rn	Vd	0	0	1	0	index_align	Rm
														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>], <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	19-16	15-12	11	10	9	8	7-4	3-0
1	1	1	1	0	1	0	0	1	D	1	0	Rn	Vd	0	1	1	0	index_align	Rm
														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>], <Rm>
```

```
if size == '11' then SEE "VLD3 (single 3-element structure to all lanes)";
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`.
- 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	0	index_align	Rm
														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>], <Rm>
```

```
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<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`.
- 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	1	D	1	0	Rn				Vd				0	0	1	0	index_align				Rm			
																				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>], <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.

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	0	index_align				Rm			
																				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>], <Rm>
```

```
if size == '11' then SEE "VLD3 (single 3-element structure to all lanes)";
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`.
- 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	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	1	0	index_align	Rm
														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>], <Rm>
```

```
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<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`.
- 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”:

size	<size>
00	8
01	16
10	32

<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];
    Elem[D[d], index,8*ebytes] = MemU[address,ebytes];
    Elem[D[d2],index,8*ebytes] = MemU[address+ebytes,ebytes];
    Elem[D[d3],index,8*ebytes] = 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;
```

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_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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	19-16	15-12	11	10	9	8	7-6	5-4	3-0
1	1	1	1	0	1	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>
```

```
integer inc;
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	3-0	15-12	11	10	9	8	7-6	5-4	3-0
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>
```

```
integer inc;
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.

For more information about the CONSTRAINED 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”:

size	<size>
00	8
01	16
10	32
11	RESERVED

<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];

    boolean nontemporal = FALSE;
    boolean tagchecked  = FALSE;
    AccessDescriptor accdesc = CreateAccDescASIMD(MemOp_LOAD, nontemporal, tagchecked);
    if !IsAligned(address, alignment) then
        AArch32.Abort(address, AlignmentFault(accdesc));

    for e = 0 to elements-1
        Elem[D[d], e,8*ebytes] = MemU[address,ebytes];
        Elem[D[d2],e,8*ebytes] = MemU[address+ebytes,ebytes];
        Elem[D[d3],e,8*ebytes] = MemU[address+2*ebytes,ebytes];
        Elem[D[d4],e,8*ebytes] = 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;
```

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_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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

31	30	29	28	27	26	25	24	23	22	21	20	19-16	15-12	11	10	9	8	7-6	5	4	3-0
1	1	1	1	0	1	0	0	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;
integer ebytes;
integer alignment;
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.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3-0	15-12	11	10	9	8	7-6	5	4	3-0
1	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;
integer ebytes;
integer alignment;
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":

size	<size>
00	8
01	16
1x	32

<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 <align> is present, the permitted values and encoding depend on <size>:

<size> == 8

<align> is 32, meaning 32-bit alignment, encoded in the "a" field as 1.

<size> == 16

<align> is 64, meaning 64-bit alignment, encoded in the "a" field as 1.

<size> == 32

<align> 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 <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];

    boolean nontemporal = FALSE;
    boolean tagchecked  = FALSE;
    AccessDescriptor accdesc = CreateAccDescASIMD(MemOp_LOAD, nontemporal, tagchecked);
    if !IsAligned(address, alignment) then
        AArch32.Abort(address, AlignmentFault(accdesc));

    constant integer esize = ebytes * 8;
    bits(esize) element1 = MemU[address, ebytes];
    bits(esize) element2 = MemU[address+ebytes,ebytes];
    bits(esize) element3 = MemU[address+2*ebytes,ebytes];
    bits(esize) element4 = MemU[address+3*ebytes,ebytes];
    D[d] = Replicate(element1, 64 DIV esize);
    D[d2] = Replicate(element2, 64 DIV esize);
    D[d3] = Replicate(element3, 64 DIV esize);
    D[d4] = Replicate(element4, 64 DIV esize);
    if wback then
        if register_index then
            R[n] = R[n] + R[m];
        else
            R[n] = R[n] + 4*ebytes;
```

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

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

31	30	29	28	27	26	25	24	23	22	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	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)";
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

31	30	29	28	27	26	25	24	23	22	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	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)";
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.
- 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	19-16	15-12	11	10	9	8	7-4	3-0
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 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.

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	D	1	0	Rn				Vd				0	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)";
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.

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			
																				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);
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	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	1	1	index_align	Rm
														size	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 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:

{ <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];

    boolean nontemporal = FALSE;
    boolean tagchecked  = FALSE;
    AccessDescriptor accdesc = CreateAccDescASIMD(MemOp_LOAD, nontemporal, tagchecked);
    if !IsAligned(address, alignment) then
        AArch32.Abort(address, AlignmentFault(accdesc));

    Elem[D[d], index,8*ebytes] = MemU[address,ebytes];
    Elem[D[d2],index,8*ebytes] = MemU[address+ebytes,ebytes];
    Elem[D[d3],index,8*ebytes] = MemU[address+2*ebytes,ebytes];
    Elem[D[d4],index,8*ebytes] = 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;
```

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_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

VLDM, VLDMDB, VLDMIA

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

31 30 29 28	27	26	25	24	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	1	imm8<7:1>	0
cond																imm8<0>

Decrement Before (P == 1 && U == 0 && W == 1)

```
VLDMDB{<c>}{<q>}{.<size>} <Rn>!, <dreglist>
```

Increment After (P == 0 && U == 1)

```
VLDM{<c>}{<q>}{.<size>} <Rn>{!}, <dreglist>

VLDMIA{<c>}{<q>}{.<size>} <Rn>{!}, <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 `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 `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															

Decrement Before (P == 1 && U == 0 && W == 1)

```
VLDMDB{<c>}{<q>}{.<size>} <Rn>!, <sreglist>
```

Increment After (P == 0 && U == 1)

```
VLDM{<c>}{<q>}{.<size>} <Rn>{!}, <sreglist>
```

```
VLDMIA{<c>}{<q>}{.<size>} <Rn>{!}, <sreglist>
```

```
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 = 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 VLDM with the same addressing mode but loads no registers.

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

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	P	U	D	W	1	Rn	Vd	1 0	1 1	imm8<7:1>	0
																	imm8<0>

Decrement Before (P == 1 && U == 0 && W == 1)

```
VLDMDB{<c>}{<q>}{.<size>} <Rn>!, <dreglist>
```

Increment After (P == 0 && U == 1)

```
VLDM{<c>}{<q>}{.<size>} <Rn>{!}, <dreglist>
```

```
VLDMIA{<c>}{<q>}{.<size>} <Rn>{!}, <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 `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 `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.

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 0 1 1 0	P	U	D	W	1	Rn	Vd	1 0	1 0	imm8

Decrement Before (P == 1 && U == 0 && W == 1)

```
VLDMDB{<c>}{<q>}{.<size>} <Rn>!, <sreglist>
```

Increment After (P == 0 && U == 1)

```
VLDM{<c>}{<q>}{.<size>} <Rn>{!}, <sreglist>
```

```
VLDMIA{<c>}{<q>}{.<size>} <Rn>{!}, <sreglist>
```

```
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 = 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 VLDM with the same addressing mode but loads no registers.

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 *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*.
<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.
!	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

Alias	Is preferred when
VPOP	`P == '0' && U == '1' && W == '1' && Rn == '1101'`

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(AccessType_ASIMD) then word1:word2 else word2:word1;

    if wback then R[n] = if add then R[n]+imm32 else R[n]-imm32;
```

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_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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

31 30 29 28	27	26	25	24	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	1	U	D	0	1	!= 1111	Vd	1	0	size	imm8
cond									Rn					

Half-precision scalar (size == 01)
(FEAT_FP16)

```
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);
integer d;
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

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	1	U	D	0	1	!= 1111	Vd	1	0	size	imm8
												Rn					

Half-precision scalar (size == 01)
(FEAT_FP16)

```
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);
integer d;
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' && 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”:

U	+/-
0	-
1	+

<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.

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(AccessType_ASIMD) then word1:word2 else word2:word1;
```

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_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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

31 30 29 28	27	26	25	24	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	1	U	D	0	1	1 1 1 1	Vd	1	0	size	imm8
cond									Rn					

Half-precision scalar (size == 01)
(FEAT_FP16)

```
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);
integer d;
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

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	1	U	D	0	1	1 1 1 1	Vd	1	0	size	imm8
												Rn					

Half-precision scalar (size == 01)
(FEAT_FP16)

```
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);
integer d;
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' && 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”:

U	+/-
0	-
1	+

<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.

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();
    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(AccessType_ASIMD) then word1:word2 else word2:word1;
```

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_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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

31	30	29	28	27	26	25	24	23	22	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	0	Vm			
										op																					

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');
integer esize;
integer elements;
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	0	sz	Vn				Vd				1	1	1	1	N	Q	M	0	Vm			
										op																					

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');
integer esize;
integer elements;
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”:

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.

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

```
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_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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
																				op	

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 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);  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	U	1	1	1	1	0	D	size	Vn	Vd	0	1	1	0	N	Q	M	0	Vm
																				op	

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 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);  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”:

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.

<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 = 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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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

31	30	29	28	27	26	25	24	23	22	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	Q	M	1	Vm			
										op																					

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;
integer esize;
integer elements;
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	0	1	D	0	0	Vn				Vd				1	0	!= 00		N	0	M	0	Vm			
																						size			op						

Half-precision scalar (size == 01)
(FEAT_FP16)

```
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');
integer esize;
integer d;
integer n;
integer m;
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

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	1	N	Q	M	1	Vm			
										op																					

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 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;
integer esize;
integer elements;
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

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	1	D	0	0	Vn	Vd	1	0	!= 00	N	0	M	0	Vm
																size		op			

Half-precision scalar (size == 01)
(FEAT_FP16)

```
VMAXNM{<q>}.F16 <Sd>, <Sn>, <Sm> // (Not permitted in IT block)
```

Single-precision scalar (size == 10)

```
VMAXNM{<q>}.F32 <Sd>, <Sn>, <Sm> // (Not permitted in IT block)
```

Double-precision scalar (size == 11)

```
VMAXNM{<q>}.F64 <Dd>, <Dn>, <Dm> // (Not permitted in IT block)
```

```
if InITBlock() then UNPREDICTABLE;
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
advsimd = FALSE;
maximum = (op == '0');
integer esize;
integer d;
integer n;
integer m;
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

```
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_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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			
										op																					

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');
integer esize;
integer elements;
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			
										op																					

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');
integer esize;
integer elements;
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”:

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.

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

```
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_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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)

```
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 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);  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	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)

```
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 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);  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”:

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.

<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 = 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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

VMINNM

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
				op									

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;
integer esize;
integer elements;
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 0 1	D	0 0	Vn	Vd	1 0	!= 00	N	1	M	0	Vm
						size		op			

Half-precision scalar (size == 01)
(FEAT_FP16)

```
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');
integer esize;
integer d;
integer n;
integer m;
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

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	1	sz	Vn				Vd				1	1	1	1	N	Q	M	1	Vm			
										op																					

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 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;
integer esize;
integer elements;
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

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	1	D	0	0	Vn				Vd				1	0	!= 00		N	1	M	0	Vm			
																						size			op						

Half-precision scalar (size == 01)
(FEAT_FP16)

```
VMINNM{<q>}.F16 <Sd>, <Sn>, <Sm> // (Not permitted in IT block)
```

Single-precision scalar (size == 10)

```
VMINNM{<q>}.F32 <Sd>, <Sn>, <Sm> // (Not permitted in IT block)
```

Double-precision scalar (size == 11)

```
VMINNM{<q>}.F64 <Dd>, <Dn>, <Dm> // (Not permitted in IT block)
```

```
if InITBlock() then UNPREDICTABLE;
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
advsimd = FALSE;
maximum = (op == '0');
integer esize;
integer d;
integer n;
integer m;
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

```
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_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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
										size				op							

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<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;
integer esize;
integer elements;
integer m;
integer index;
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	Q	1	1	1	1	1	D	!= 11	Vn	Vd	0	0	0	F	N	1	M	0	Vm
										size				op							

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 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;
integer esize;
integer elements;
integer m;
integer index;
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”:

F	size	<dt>
0	01	I16
0	10	I32
1	01	F16
1	10	F32

<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
                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[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 and is operating only on integer vector elements, then the following apply:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to 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_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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

31 30 29 28 27 26 25	24	23	22	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 0 1	N	Q	M	1	Vm
				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;
advsimd = TRUE;  add = (op == '0');
integer esize;
integer elements;
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	0 0	Vn	Vd	1 0	size	N	0	M	0	Vm
cond										op			

Half-precision scalar (size == 01)
(FEAT_FP16)

```
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');
integer esize;
integer d;
integer n;
integer m;
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	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	sz	Vn				Vd				1	1	0	1	N	Q	M	1	Vm			
										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');
integer esize;
integer elements;
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	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	D	0	0	Vn				Vd				1	0	size		N	0	M	0	Vm			
																									op						

Half-precision scalar (size == 01)
(FEAT_FP16)

```
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');
integer esize;
integer d;
integer n;
integer m;
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”:

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
                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(D[n], D[m], FPSCR[])
                            else FPNeg(FPMul(D[n], D[m], FPSCR[])));
                D[d] = FPAdd(D[d], addend64, FPSCR[]);
```

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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

31–25	24	23	22	21–20	19–16	15–12	11–8	7	6	5	4	3–0
1 1 1 1 0 0 1	0	0	D	size	Vn	Vd	1 0 0 1	N	Q	M	0	Vm
	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 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

15–13	12	11–7	6	5–4	3–0	15–12	11–8	7	6	5	4	3–0
1 1 1	0	1 1 1 1 0	D	size	Vn	Vd	1 0 0 1	N	Q	M	0	Vm
	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 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*.

<dt> Is the data type for the elements of the operands, encoded in “size”:

size	<dt>
00	I8
01	I16
10	I32

<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
            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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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

31	30	29	28	27	26	25	24	23	22	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	0	N	1	M	0	Vm
										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;
integer esize;
integer elements;
integer m;
integer index;
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	U	1	1	1	1	1	D	!= 11	Vn	Vd	0	0	1	0	N	1	M	0	Vm
										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;
integer esize;
integer elements;
integer m;
integer index;
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”:

U	size	<dt>
0	01	S16
0	10	S32
1	01	U16
1	10	U32

<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[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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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>}.<dt> <Qd>, <Dn>, <Dm>
```

```
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	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>}.<dt> <Qd>, <Dn>, <Dm>
```

```
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*.

<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

```
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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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

31	30	29	28	27	26	25	24	23	22	21-20	19-16	15-12	11	10	9	8	7	6	5	4	3-0
1	1	1	1	0	0	1	Q	1	D	!= 11	Vn	Vd	0	1	0	F	N	1	M	0	Vm
										size				op							

64-bit SIMD vector (Q == 0)

```
VMLS{<c>}{<q>}.<dt> <Dd>, <Dn>, <Dm[x]>
```

128-bit SIMD vector (Q == 1)

```
VMLS{<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<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;
integer esize;
integer elements;
integer m;
integer index;
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	3-0	15-12	11	10	9	8	7	6	5	4	3-0
1	1	1	Q	1	1	1	1	1	D	!= 11	Vn	Vd	0	1	0	F	N	1	M	0	Vm
										size				op							

64-bit SIMD vector (Q == 0)

```
VMLS{<c>}{<q>}.<dt> <Dd>, <Dn>, <Dm[x]>
```

128-bit SIMD vector (Q == 1)

```
VMLS{<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 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;
integer esize;
integer elements;
integer m;
integer index;
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”:

F	size	<dt>
0	01	I16
0	10	I32
1	01	F16
1	10	F32

<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
                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[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 and is operating only on integer vector elements, then the following apply:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to 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_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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.

Note

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*, *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	19-16	15-12	11	10	9	8	7	6	5	4	3-0
1	1	1	1	0	0	1	0	0	D	1	sz	Vn	Vd	1	1	0	1	N	Q	M	1	Vm
										op												

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');
integer esize;
integer elements;
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

31-28	27	26	25	24	23	22	21	20	19-16	15-12	11	10	9-8	7	6	5	4	3-0
!= 1111	1	1	1	0	0	D	0	0	Vn	Vd	1	0	size	N	1	M	0	Vm
cond															op			

Half-precision scalar (size == 01)
(FEAT_FP16)

```
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');
integer esize;
integer d;
integer n;
integer m;
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	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	0	1	N	Q	M	1	Vm			
										op																					

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;
if sz == '1' && InITBlock() then UNPREDICTABLE;
advsimd = TRUE;  add = (op == '0');
integer esize;
integer elements;
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	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	D	0	0	Vn				Vd				1	0	size		N	1	M	0	Vm			
																									op						

Half-precision scalar (size == 01)
(FEAT_FP16)

```
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');
integer esize;
integer d;
integer n;
integer m;
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”:

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
                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(D[n], D[m], FPSCR[])
                            else FPNeg(FPMul(D[n], D[m], FPSCR[])));
                D[d] = FPAdd(D[d], addend64, FPSCR[]);
```

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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

31	30	29	28	27	26	25	24	23	22	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	1	N	Q	M	0	Vm
							op														

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 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

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	1	N	Q	M	0	Vm
			op																		

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 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*.

<dt> Is the data type for the elements of the operands, encoded in “size”:

size	<dt>
00	I8
01	I16
10	I32

<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
            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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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	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;
integer esize;
integer elements;
integer m;
integer index;
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	U	1	1	1	1	1	D	!= 11	Vn	Vd	0	1	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;
integer esize;
integer elements;
integer m;
integer index;
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”:

U	size	<dt>
0	01	S16
0	10	S32
1	01	U16
1	10	U32

<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[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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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
1	1	1	1	0	0	1	U	1	D	!= 11	Vn	Vd	1	0	1	0	N	0	M	0	Vm
										size					op						

A1

```
VMLSL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>
```

```
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	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	1	0	N	0	M	0	Vm
										size					op						

T1

```
VMLSL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>
```

```
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*.

<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

```
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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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.

Note

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
(FEAT_AA32BF16)

31	30	29	28	27	26	25	24	23	22	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
(FEAT_AA32BF16)

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	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 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

(FEAT_FP16)

31 30 29 28	27	26	25	24	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)	1	(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

(FEAT_FP16)

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	0	0	0	op	Vn	Rt	1	0	0	1	N	(0)	(0)	1	(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
```

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

<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] = Zeros(16) : S[n]<15:0>;
    else
        S[n] = Zeros(16) : R[t]<15:0>;
```

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)	1	(0)	(0)	(0)	(0)
cond																						

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	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)	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];
```

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_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9	8	7	6	5	4	3–0
!= 1111	1	1	0	0	0	1	0	op	Rt2	Rt	1	0	1	1	0	0	M	1	Vm
cond																			

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	3–0	15–12	11	10	9	8	7	6	5	4	3–0
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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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*, *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	op	Rt2	Rt	1	0	1	0	0	0	M	1	Vm
cond																			

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

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	op	Rt2	Rt	1	0	1	0	0	0	M	1	Vm

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.

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
    EncodingSpecificOperations();  CheckVFPEnabled(TRUE);
    if to_arm_registers then
        R[t] = S[m];
        R[t2] = S[m+1];
    else
        S[m] = R[t];
        S[m+1] = 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 (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	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>
```

```
boolean advsimd;
integer esize;
integer index;
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	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	opc1	0	Vd	Rt	1	0	1	1	D	opc2	1	(0)	(0)	(0)	(0)

T1

```
VMOV{<c>}{<q>}{.<size>} <Dd[x]>, <Rt>
```

```
boolean advsimd;
integer esize;
integer index;
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.

16
Encoded as opc1<1> = 0, opc2<0> = 1. [x] is encoded in opc1<0>, opc2<1>.

32
Encoded as opc1<1> = 0, opc2 = 0b00. [x] is encoded in opc1<0>.

omitted
Equivalent to 32.

<Dd[x]> The scalar. The register <Dd> is encoded in D:Vd. For details of how [x] is encoded, see the description of <size>.

<Rt> 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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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

31	30	29	28	27	26	25	24	23	22	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	i	1	D	0	0	0	imm3			Vd				0	x	x	0	0	Q	0	1	imm4			
																				cmode						op					

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

31	30	29	28	27	26	25	24	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	imm4H				Vd				1	0	size		(0)	0	(0)	0	imm4L			
cond																															

Half-precision scalar (size == 01)
(FEAT_FP16)

```
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;
integer d;
integer regs;
case size of
    when '01' d = UInt(Vd:D);  imm16 = VFPExpandImm(imm4H:imm4L, 16); imm32 = Zeros(16) : imm16;
    when '10' d = UInt(Vd:D);  imm32 = VFPExpandImm(imm4H:imm4L, 32);
    when '11' d = UInt(D:Vd);  imm64 = VFPExpandImm(imm4H:imm4L, 64);  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

31	30	29	28	27	26	25	24	23	22	21	20	19	18-16	15-12	11	10	9	8	7	6	5	4	3-0
1	1	1	1	0	0	1	i	1	D	0	0	0	imm3	Vd	1	0	x	0	0	Q	0	1	imm4
															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>
```

```
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;
```

A4

31	30	29	28	27	26	25	24	23	22	21	20	19	18-16	15-12	11	10	9	8	7	6	5	4	3-0
1	1	1	1	0	0	1	i	1	D	0	0	0	imm3	Vd	1	1	x	x	0	Q	0	1	imm4
															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>
```

```
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;
```

A5

31	30	29	28	27	26	25	24	23	22	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	i	1	D	0	0	0	imm3			Vd				1	1	1	0	0	Q	1	1	imm4			
																				cmode						op					

64-bit SIMD vector (Q == 0)

```
VMOV{<c>}{<q>}.I64 <Dd>, #<imm>
```

128-bit SIMD vector (Q == 1)

```
VMOV{<c>}{<q>}.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;
```

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	i	1	1	1	1	1	D	0	0	0	imm3			Vd				0	x	x	0	0	Q	0	1	imm4			
																				cmode						op					

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;
```

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	0	1	1	1	0	1	D	1	1	imm4H				Vd				1	0	size		(0)	0	(0)	0	imm4L			

Half-precision scalar (size == 01)
(FEAT_FP16)

```
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;
integer d;
integer regs;
case size of
    when '01' d = UInt(Vd:D);  imm16 = VFPExpandImm(imm4H:imm4L, 16); imm32 = Zeros(16) : imm16;
    when '10' d = UInt(Vd:D);  imm32 = VFPExpandImm(imm4H:imm4L, 32);
    when '11' d = UInt(D:Vd);  imm64 = VFPExpandImm(imm4H:imm4L, 64);  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

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	i	1	1	1	1	1	D	0	0	0	imm3			Vd				1	0	x	0	0	Q	0	1	imm4			
																				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>
```

```
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;
```

T4

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	i	1	1	1	1	1	D	0	0	0	imm3			Vd				1	1	x	x	0	Q	0	1	imm4			
																				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>
```

```
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	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			
																				cmode						op					

64-bit SIMD vector (Q == 0)

```
VMOV{<c>}{<q>}.I64 <Dd>, #<imm>
```

128-bit SIMD vector (Q == 1)

```
VMOV{<c>}{<q>}.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.

For encoding A2, T1, T2, T3, T4 and T5: see *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

<dt> The data type, encoded in “cmode”:

cmode	<dt>
110x	I32
1110	I8
1111	F32

<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.

<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
        S[d] = imm32;
    else
        for r = 0 to regs-1
            D[d+r] = 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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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>
```

```
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
single_register = (size == '10');  advsimd = FALSE;
integer d;
integer m;
integer regs;
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	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	0	0	0	Vd				1	0	1	x	0	1	M	0	Vm			
																						size									

Single-precision scalar (size == 10)

```
VMOV{<c>}{<q>}.F32 <Sd>, <Sm>
```

Double-precision scalar (size == 11)

```
VMOV{<c>}{<q>}.F64 <Dd>, <Dm>
```

```
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
single_register = (size == '10');  advsimd = FALSE;
integer d;
integer m;
integer regs;
if single_register then
    d = UInt(Vd:D);  m = UInt(Vm:M);
else
    d = UInt(D:Vd);  m = UInt(M:Vm);  regs = 1;
```

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();  CheckAdvSIMDOrVFPEnabled(TRUE, advsimd);
    if single_register then
        S[d] = S[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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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, any CONSTRAINED UNPREDICTABLE behavior, and any operational information 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	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	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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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				1	1	1	0	U	opc1		1	Vn				Rt				1	0	1	1	N	opc2		1	(0)	(0)	(0)	(0)
cond																															

A1

```
VMOV{<c>}{<q>}{.<dt>} <Rt>, <Dn[x]>
```

```
boolean advsimd;
integer esize;
integer index;
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	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	0	U	opc1		1	Vn				Rt				1	0	1	1	N	opc2		1	(0)	(0)	(0)	(0)

T1

```
VMOV{<c>}{<q>}{.<dt>} <Rt>, <Dn[x]>
```

```
boolean advsimd;
integer esize;
integer index;
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

```
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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

VMOVL

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 *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	!= 000			0	0	0	Vd				1	0	1	0	0	0	M	1	Vm			
										imm3H																					

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

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	!= 000			0	0	0	Vd				1	0	1	0	0	0	M	1	Vm			
										imm3H																					

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 *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*.

<dt> Is the data type for the elements of the operand, encoded in “U:imm3H”:

U	imm3H	<dt>
0	001	S8
0	010	S16
0	100	S32
1	001	U8
1	010	U16
1	100	U32

<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

```
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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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

31	30	29	28	27	26	25	24	23	22	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	0	0	M	0	Vm			

A1

```
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

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	0	M	0	Vm			

T1

```
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”:

size	<dt>
00	I16
01	I32
10	I64
11	RESERVED

<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
        Elem[D[d],e,esize] = 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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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
(FEAT_FP16)

31	30	29	28	27	26	25	24	23	22	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
(FEAT_FP16)

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	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>;
```

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 Advanced SIMD and floating-point 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

31	30	29	28	27	26	25	24	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	1	reg				Rt				1	0	1	0	(0)	(0)	(0)	1	(0)	(0)	(0)	(0)
cond																															

A1

```
VMRS{<c>}{<q>} <Rt>, <spec_reg>
```

```
t = UInt(Rt);
if !(reg IN {'000x', '0101', '011x', '1000'}) then UNPREDICTABLE;
if t == 15 && reg != '0001' then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

CONSTRAINED UNPREDICTABLE behavior

If `!(reg IN {'000x', '0101', '011x', '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

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	1	1	1	1	reg				Rt				1	0	1	0	(0)	(0)	(0)	1	(0)	(0)	(0)	(0)

T1

```
VMRS{<c>}{<q>} <Rt>, <spec_reg>
```

```
t = UInt(Rt);
if !(reg IN {'000x', '0101', '011x', '1000'}) then UNPREDICTABLE;
if t == 15 && reg != '0001' then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

CONSTRAINED UNPREDICTABLE behavior

If `!(reg IN {'000x', '0101', '011x', '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”:

reg	<spec_reg>
0000	FPSID
0001	FPSCR
001x	UNPREDICTABLE
0100	UNPREDICTABLE
0101	MVFR2
0110	MVFR1
0111	MVFR0
1000	FPEXC
1001	UNPREDICTABLE
101x	UNPREDICTABLE
11xx	UNPREDICTABLE

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if reg == '0001' then                 // FPSCR
        CheckVFPEnabled(TRUE);
        if t == 15 then
            PSTATE.<N,Z,C,V> = FPSR.<N,Z,C,V>;
        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 Advanced SIMD and floating-point 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																															

A1

```
VMSR{<c>}{<q>} <spec_reg>, <Rt>
```

```
t = UInt(Rt);
if !(reg IN {'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	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	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 IN {'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”:

reg	<spec_reg>
0000	FPSID
0001	FPSCR
001x	UNPREDICTABLE
01xx	UNPREDICTABLE
1000	FPEXC
1001	UNPREDICTABLE
101x	UNPREDICTABLE
11xx	UNPREDICTABLE

<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
            when '1000'
                FPEXC = R[t];
            otherwise
                Unreachable();   // Dealt with above or in encoding-specific pseudocode
```

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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	19-16	15-12	11	10	9	8	7	6	5	4	3-0
1	1	1	1	0	0	1	Q	1	D	!= 11	Vn	Vd	1	0	0	F	N	1	M	0	Vm
										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>]
```

```
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
floating_point = (F == '1');  long_destination = FALSE;
d = UInt(D:Vd);  n = UInt(N:Vn);  regs = if Q == '0' then 1 else 2;
integer esize;
integer elements;
integer m;
integer index;
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	3-0	15-12	11	10	9	8	7	6	5	4	3-0
1	1	1	Q	1	1	1	1	1	D	!= 11	Vn	Vd	1	0	0	F	N	1	M	0	Vm
										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>]
```

```
if size == '11' then SEE "Related encodings";
if F == '1' && size == '01' && InITBlock() then UNPREDICTABLE;
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
floating_point = (F == '1');  long_destination = FALSE;
d = UInt(D:Vd);  n = UInt(N:Vn);  regs = if Q == '0' then 1 else 2;
integer esize;
integer elements;
integer m;
integer index;
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”:

F	size	<dt>
0	01	I16
0	10	I32
1	01	F16
1	10	F32

<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>>1],e,2*esize] = (op1val*op2val)<2*esize-1:0>;
                else
                    Elem[D[d+r],e,esize] = (op1val*op2val)<esize-1:0>;
```

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	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 Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
advsimd = TRUE;
integer esize;
integer elements;
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)
(FEAT_FP16)

```
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;
integer esize;
integer d;
integer n;
integer m;
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	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	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;
integer esize;
integer elements;
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	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	D	1	0	Vn				Vd				1	0	size		N	0	M	0	Vm			

Half-precision scalar (size == 01)
(FEAT_FP16)

```
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' && InITBlock()  then UNPREDICTABLE;
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
advsimd = FALSE;
integer esize;
integer d;
integer n;
integer m;
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”:

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
                Elem[D[d+r],e,esize] = FPMul(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) : 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[]);
```

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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 *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	op	0	D	size		Vn				Vd				1	0	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 size == '11' || (op == '1' && size != '00') then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
unsigned = FALSE;  // "Don't care" value: TRUE produces same functionality
polynomial = (op == '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	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	op	1	1	1	1	0	D	size		Vn				Vd				1	0	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 size == '11' || (op == '1' && size != '00') then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
unsigned = FALSE;  // "Don't care" value: TRUE produces same functionality
polynomial = (op == '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 “op:size”:

op	size	<dt>
0	00	I8
0	01	I16
0	10	I32
1	00	P8

<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 = Elem[Din[n+r],e,esize];  op1val = Int(op1, unsigned);
            op2 = Elem[Din[m+r],e,esize];  op2val = Int(op2, unsigned);
            bits(2 * esize) product;
            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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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;
integer esize;
integer elements;
integer index;
integer m;
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	U	1	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;
integer esize;
integer elements;
integer index;
integer m;
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”:

U	size	<dt>
0	01	S16
0	10	S32
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 "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_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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
										size											

A1

```
VMULL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>
```

```
if size == '11' then SEE "Related encodings";
unsigned = (U == '1'); polynomial = (op == '1'); long_destination = TRUE;
esize = 8 << UInt(size); elements = 64 DIV esize;
if polynomial then
    if U == '1' || size == '01' then UNDEFINED;
    if size == '10' then    // .p64
        if !HaveBit128PMULLExt() then UNDEFINED;
        esize = 64; elements = 1;
if Vd<0> == '1' then UNDEFINED;
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	D	!= 11	Vn	Vd	1	1	op	0	N	0	M	0	Vm
										size											

T1

```
VMULL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>
```

```
if size == '11' then SEE "Related encodings";
unsigned = (U == '1'); polynomial = (op == '1'); long_destination = TRUE;
esize = 8 << UInt(size); elements = 64 DIV esize;
if polynomial then
    if U == '1' || size == '01' then UNDEFINED;
    if size == '10' then    // .p64
        if InITBlock() then UNPREDICTABLE;
        if !HaveBit128PMULLExt() then UNDEFINED;
        esize = 64; elements = 1;
if Vd<0> == '1' then UNDEFINED;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); regs = 1;
```

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

<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:U:size”:

op	U	size	<dt>
0	0	00	S8
0	0	01	S16
0	0	10	S32
0	1	00	U8
0	1	01	U16
0	1	10	U32
1	0	00	P8
1	0	10	P64

<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);
            bits(2 * esize) product;
            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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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

31	30	29	28	27	26	25	24	23	22	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	i	1	D	0	0	0	imm3	Vd	0	x	x	0	0	Q	1	1	imm4
															cmode								

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;
```

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	i	1	D	0	0	0	imm3	Vd	1	0	x	0	0	Q	1	1	imm4
															cmode								

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;
```

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	0	1	i	1	D	0	0	0	imm3	Vd	1	1	0	x	0	Q	1	1	imm4
															cmode								

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;
```

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	i	1	1	1	1	1	D	0	0	0	imm3			Vd				0	x	x	0	0	Q	1	1	imm4			
																				cmode											

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;
```

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	i	1	1	1	1	1	D	0	0	0	imm3			Vd				1	0	x	0	0	Q	1	1	imm4			
																				cmode											

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;
```

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	i	1	1	1	1	1	D	0	0	0	imm3			Vd				1	1	0	x	0	Q	1	1	imm4			
																				cmode											

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;
```

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();  CheckAdvSIMDEnabled();
    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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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

31	30	29	28	27	26	25	24	23	22	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	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

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		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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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*, *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	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> <Qd>, <Qm>
```

```
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

31	30	29	28	27	26	25	24	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	1	Vd				1	0	size		0	1	M	0	Vm			
cond																															

Half-precision scalar (size == 01)
(FEAT_FP16)

```
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;
integer esize;
integer d;
integer m;
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	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	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> <Qd>, <Qm>
```

```
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	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	0	0	1	Vd				1	0	size		0	1	M	0	Vm			

Half-precision scalar (size == 01)
(FEAT_FP16)

```
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' && InITBlock()  then UNPREDICTABLE;
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
advsimd = FALSE;
integer esize;
integer d;
integer m;
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

<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”:

F	size	<dt>
0	00	S8
0	01	S16
0	10	S32
1	01	F16
1	10	F32

<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

```
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
                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>;
    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]);
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check and is operating only on integer vector elements, then the following apply:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to 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_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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.

Note

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*, *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	D	0	1	Vn	Vd	1	0	size	N	1	M	0	Vm
cond															op			

Half-precision scalar (size == 01)
(FEAT_FP16)

```
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;
integer esize;
integer d;
integer n;
integer m;
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	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	D	0	1	Vn				Vd				1	0	size		N	1	M	0	Vm			
																									op						

Half-precision scalar (size == 01)
(FEAT_FP16)

```
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;
integer esize;
integer d;
integer n;
integer m;
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
            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);
```

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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*, *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	D	0	1	Vn	Vd	1	0	size	N	0	M	0	Vm
cond															op			

Half-precision scalar (size == 01)
(FEAT_FP16)

```
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;
integer esize;
integer d;
integer n;
integer m;
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	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	D	0	1	Vn	Vd	1	0	size	N	0	M	0	Vm
																		op			

Half-precision scalar (size == 01)
(FEAT_FP16)

```
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;
integer esize;
integer d;
integer n;
integer m;
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
            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);
```

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

VNMUL

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

31 30 29 28	27	26	25	24	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	1	M	0	Vm
cond																		

Half-precision scalar (size == 01)
(FEAT_FP16)

```
VNMUL{<c>}{<q>}.F16 {<Sd>,} <Sn>, <Sm>
```

Single-precision scalar (size == 10)

```
VNMUL{<c>}{<q>}.F32 {<Sd>,} <Sn>, <Sm>
```

Double-precision scalar (size == 11)

```
VNMUL{<c>}{<q>}.F64 {<Dd>,} <Dn>, <Dm>
```

```
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;
integer esize;
integer d;
integer n;
integer m;
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	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	D	1	0	Vn	Vd	1	0	size	N	1	M	0	Vm

Half-precision scalar (size == 01)
(FEAT_FP16)

```
VNMUL{<c>}{<q>}.F16 {<Sd>,} <Sn>, <Sm>
```

Single-precision scalar (size == 10)

```
VNMUL{<c>}{<q>}.F32 {<Sd>,} <Sn>, <Sm>
```

Double-precision scalar (size == 11)

```
VNMUL{<c>}{<q>}.F64 {<Dd>,} <Dn>, <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;
integer esize;
integer d;
integer n;
integer m;
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
            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);
```

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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, any CONSTRAINED UNPREDICTABLE behavior, and any operational information 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	i	1	D	0	0	0	imm3			Vd				0	x	x	1	0	Q	0	1	imm4			
																				cmode											

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

31	30	29	28	27	26	25	24	23	22	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	i	1	D	0	0	0	imm3			Vd				1	0	x	1	0	Q	0	1	imm4			
																				cmode											

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

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	i	1	1	1	1	1	D	0	0	0	imm3			Vd				0	x	x	1	0	Q	0	1	imm4			
																				cmode											

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

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	i	1	1	1	1	1	D	0	0	0	imm3			Vd				1	0	x	1	0	Q	0	1	imm4			
																				cmode											

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.

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

31	30	29	28	27	26	25	24	23	22	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				0	0	0	1	N	Q	M	1	Vm			

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;
```

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	1	Vn				Vd				0	0	0	1	N	Q	M	1	Vm			

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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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

31	30	29	28	27	26	25	24	23	22	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	i	1	D	0	0	0	imm3			Vd				0	x	x	1	0	Q	0	1	imm4			
																				cmode											

64-bit SIMD vector (Q == 0)

```
VORR{<c>}{<q>}.I32 {<Dd>,} <Dd>, #<imm>
```

128-bit SIMD vector (Q == 1)

```
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

31	30	29	28	27	26	25	24	23	22	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	i	1	D	0	0	0	imm3			Vd				1	0	x	1	0	Q	0	1	imm4			
																				cmode											

64-bit SIMD vector (Q == 0)

```
VORR{<c>}{<q>}.I16 {<Dd>,} <Dd>, #<imm>
```

128-bit SIMD vector (Q == 1)

```
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

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	i	1	1	1	1	1	D	0	0	0	imm3			Vd				0	x	x	1	0	Q	0	1	imm4			
																				cmode											

64-bit SIMD vector (Q == 0)

```
VORR{<c>}{<q>}.I32 {<Dd>,} <Dd>, #<imm>
```

128-bit SIMD vector (Q == 1)

```
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;
```

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	i	1	1	1	1	1	D	0	0	0	imm3			Vd				1	0	x	1	0	Q	0	1	imm4			
																				cmode											

64-bit SIMD vector (Q == 0)

```
VORR{<c>}{<q>}.I16 {<Dd>,} <Dd>, #<imm>
```

128-bit SIMD vector (Q == 1)

```
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;
```

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] 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

31	30	29	28	27	26	25	24	23	22	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				0	0	0	1	N	Q	M	1	Vm			

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 = 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	0	Vn				Vd				0	0	0	1	N	Q	M	1	Vm			

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 = 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.

Alias Conditions

Alias	Is preferred when
VMOV (register, SIMD)	`N:Vn == M:Vm`
VRSHR (zero)	`Never`
VSHR (zero)	`Never`

Operation

```
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.

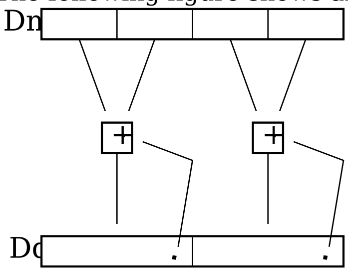
Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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	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	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

<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”:

op	size	<dt>
0	00	S8
0	01	S16
0	10	S32
0	11	RESERVED
1	00	U8
1	01	U16
1	10	U32
1	11	RESERVED

<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] = 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

31	30	29	28	27	26	25	24	23	22	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	0	1	N	Q	M	0	Vm			

A1

```
VPADD{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

```
if Q == '1' then UNDEFINED;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
integer esize;
integer elements;
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	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> {<Dd>, }<Dn>, <Dm>
```

```
if Q == '1' then UNDEFINED;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
if sz == '1' && InITBlock() then UNPREDICTABLE;
integer esize;
integer elements;
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”:

sz	<dt>
0	F32
1	F16

<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
        Elem[dest,e,esize]   = FPAdd(Elem[D[n],2*e,esize], Elem[D[n],2*e+1,esize],
                                     StandardFPSCRValue());
        Elem[dest,e+h,esize] = FPAdd(Elem[D[m],2*e,esize], Elem[D[m],2*e+1,esize],
                                     StandardFPSCRValue());

    D[d] = dest;
```

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

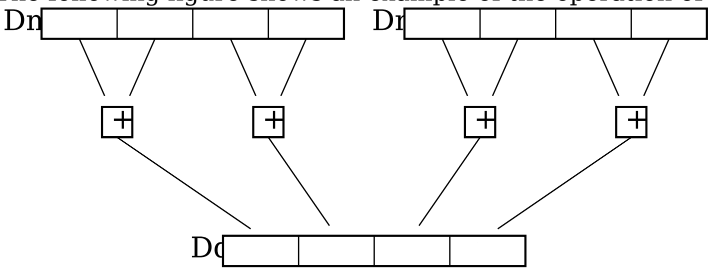
VPADD (integer)

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

31	30	29	28	27	26	25	24	23	22	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	1	1	N	Q	M	1	Vm			

A1

```
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

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	1	1	N	Q	M	1	Vm			

T1

```
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”:

size	<dt>
00	I8
01	I16
10	I32

<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
        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.

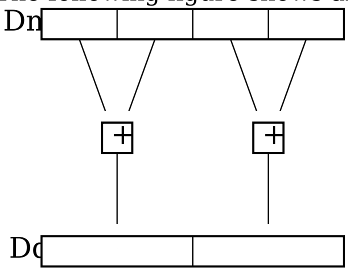
Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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.



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	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”:

op	size	<dt>
0	00	S8
0	01	S16
0	10	S32
0	11	RESERVED
1	00	U8
1	01	U16
1	10	U32
1	11	RESERVED

<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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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');
integer esize;
integer elements;
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	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	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');
integer esize;
integer elements;
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”:

sz	<dt>
0	F32
1	F16

<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.

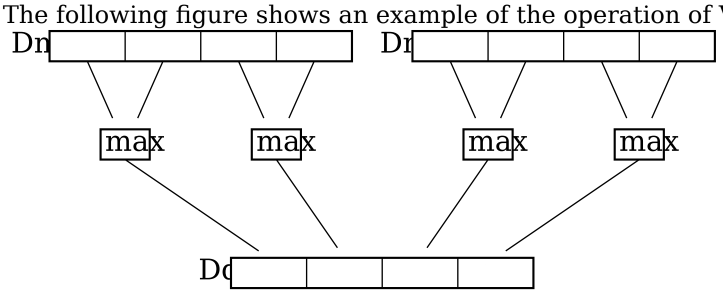
Operation

```
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;
```

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

31	30	29	28	27	26	25	24	23	22	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	1	0	1	0	N	0	M	0	Vm
																				op	

A1

```
VPMAX{<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

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	1	0	1	0	N	0	M	0	Vm
																				op	

T1

```
VPMAX{<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.

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

<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.

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

31	30	29	28	27	26	25	24	23	22	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	0	M	0	Vm			
										op																					

A1

```
VPMIN{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

```
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
maximum = (op == '0');
integer esize;
integer elements;
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	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	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');
integer esize;
integer elements;
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”:

sz	<dt>
0	F32
1	F16

<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;
```

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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

31	30	29	28	27	26	25	24	23	22	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				1	0	1	0	N	0	M	1	Vm			
																											op				

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

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				1	0	1	0	N	0	M	1	Vm			
																											op				

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.

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

<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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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, any CONSTRAINED UNPREDICTABLE behavior, and any operational information 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
!= 1111	1	1	0	0	1	D	1	1	1 1 0 1	Vd	1 0	1 1	imm8<7:1>	0
cond				P	U		W		Rn					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

31 30 29 28	27	26	25	24	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	1	D	1	1	1 1 0 1	Vd	1 0	1 0	imm8
cond				P	U		W		Rn				

Increment After

```
VPOP{<c>}{<q>}{.<size>} <sreglist>
```

is equivalent to

```
VLDM{<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	15 14 13 12	11 10	9 8	7 6 5 4 3 2 1	0
1 1 1 0 1 1 0	0	1	D	1	1	1 1 0 1	Vd	1 0	1 1	imm8<7:1>	0
	P	U		W		Rn					imm8<0>

Increment After

```
VPOP{<c>}{<q>}{.<size>} <dreglist>
```

is equivalent to

```
VLDM{<c>}{<q>}{.<size>} SP!, <dreglist>
```

and is always the preferred disassembly.

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	0	1	1	0	0	1	D	1	1	1	1	0	1	Vd				1	0	1	0	imm8							
							P	U		W		Rn																			

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.

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_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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, any CONSTRAINED UNPREDICTABLE behavior, and any operational information 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
!= 1111	1 1 0	1	0	D	1	0	1 1 0 1	Vd	1 0	1 1	imm8<7:1>	0
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
!= 1111	1 1 0	1	0	D	1	0	1 1 0 1	Vd	1 0	1 0	imm8
cond		P	U		W		Rn				

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	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	D	1	0	1 1 0 1	Vd	1 0	1 1	imm8<7:1>	0
	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.

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	0	1	1	0	1	0	D	1	0	1	1	0	1	Vd				1	0	1	0	imm8							
							P	U		W		Rn																			

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.

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

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	D	1	1	size	0	0	Vd	0	1	1	1	0	Q	M	0	Vm

64-bit SIMD vector (Q == 0)

```
VQABS{<c>}{<q>}.<dt> <Dd>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VQABS{<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	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	1	1	0	Q	M	0	Vm

64-bit SIMD vector (Q == 0)

```
VQABS{<c>}{<q>}.<dt> <Dd>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VQABS{<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":

size	<dt>
00	S8
01	S16
10	S32
11	RESERVED

<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
            result = Abs(SInt(Elem[D[m+r],e,esize]));
            boolean sat;
            (Elem[D[d+r],e,esize], sat) = SignedSatQ(result, esize);
            if sat then FPSCR.QC = '1';
```

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

VQADD

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, *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	0	0	N	Q	M	1	Vm			

64-bit SIMD vector (Q == 0)

```
VQADD{<c>}{<q>}.<dt> {<Dd>,} <Dn>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VQADD{<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	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			

64-bit SIMD vector (Q == 0)

```
VQADD{<c>}{<q>}.<dt> {<Dd>,} <Dn>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VQADD{<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

<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”:

U	size	<dt>
0	00	S8
0	01	S16
0	10	S32
0	11	S64
1	00	U8
1	01	U16
1	10	U32
1	11	U64

<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
            sum = Int(Elem[D[n+r],e,esize], unsigned) + Int(Elem[D[m+r],e,esize], unsigned);
            boolean sat;
            (Elem[D[d+r],e,esize], sat) = SatQ(sum, esize, unsigned);
            if sat then FPSCR.QC = '1';
```

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	D	!= 11	Vn	Vd	1 0	0	1	N	0	M	0	Vm
				size				op						

A1

```
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;
```

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	0	0	1 1	N	1	M	0	Vm
				size				op						

A2

```
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);
integer esize;
integer elements;
integer m;
integer index;
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 1	D	!= 11	Vn	Vd	1 0	0	1	N	0	M	0	Vm
				size				op						

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

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	1	D	!= 11		Vn				Vd				0	0	1	1	N	1	M	0	Vm			
										size											op										

T2

```
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);
integer esize;
integer elements;
integer m;
integer index;
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 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”:

size	<dt>
01	S16
10	S32

<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.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    integer op2;
    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);
        integer result;
        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);
        boolean sat2;
        (Elem[Q[d>>1],e,2*esize], sat2) = SignedSatQ(result, 2*esize);
        if sat1 || sat2 then FPSCR.QC = '1';
```

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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

31	30	29	28	27	26	25	24	23	22	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	0	M	0	Vm
										size					op						

A1

```
VQDMLSL{<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;
```

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	0	1	1	1	N	1	M	0	Vm
										size				op							

A2

```
VQDMLSL{<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);
integer esize;
integer elements;
integer m;
integer index;
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	1	D	!= 11	Vn	Vd	1	0	1	1	N	0	M	0	Vm
										size					op						

T1

```
VQDMLSL{<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

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 1	D	!= 11	Vn	Vd	0	1	1 1	N	1	M	0	Vm
				size				op						

T2

```
VQDMLSL{<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);
integer esize;
integer elements;
integer m;
integer index;
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 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”:

size	<dt>
01	S16
10	S32

<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.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    integer op2;
    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);
        integer result;
        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);
        boolean sat2;
        (Elem[Q[d>>1],e,2*esize], sat2) = SignedSatQ(result, 2*esize);
        if sat1 || sat2 then FPSCR.QC = '1';
```

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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	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> {<Dd>, }<Dn>, <Dm>
```

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	0	0	1	Q	1	D	!= 11		Vn				Vd				1	1	0	0	N	1	M	0	Vm			
										size																					

64-bit SIMD vector (Q == 0)

```
VQDMULH{<c>}{<q>}.<dt> {<Dd>,} <Dn>, <Dm[x]>
```

128-bit SIMD vector (Q == 1)

```
VQDMULH{<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);  regs = if Q == '0' then 1 else 2;
integer esize;
integer elements;
integer m;
integer index;
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	0	D	size	Vn	Vd	1	0	1	1	N	Q	M	0	Vm

64-bit SIMD vector (Q == 0)

```
VQDMULH{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

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;
```

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	Q	1	1	1	1	1	D	!= 11	Vn	Vd	1	1	0	0	N	1	M	0	Vm
										size											

64-bit SIMD vector (Q == 0)

```
VQDMULH{<c>}{<q>}.<dt> {<Dd>,} <Dn>, <Dm[x]>
```

128-bit SIMD vector (Q == 1)

```
VQDMULH{<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);  regs = if Q == '0' then 1 else 2;
integer esize;
integer elements;
integer m;
integer index;
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 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”:

size	<dt>
01	S16
10	S32

<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.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    integer op2;
    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';
```

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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											

A1

```
VQDMULL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>
```

```
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											

A2

```
VQDMULL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm[x]>
```

```
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);
integer esize;
integer elements;
integer m;
integer index;
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	1	D	!= 11	Vn	Vd	1	1	0	1	N	0	M	0	Vm
										size											

T1

```
VQDMULL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>
```

```
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

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 1	D	!= 11	Vn	Vd	1 0 1 1	N	1	M	0	Vm
				size								

T2

```
VQDMULL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm[x]>
```

```
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);
integer esize;
integer elements;
integer m;
integer index;
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 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”:

size	<dt>
01	S16
10	S32

<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[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

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    integer op2;
    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, sat) = SignedSatQ(2*op1*op2, 2*esize);
        Elem[Q[d>>1],e,2*esize] = product;
        if sat then FPSCR.QC = '1';
```

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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

31	30	29	28	27	26	25	24	23	22	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	op		M	0	Vm			

Signed result (op == 1x)

```
VQMOVN{<c>}{<q>}.<dt> <Dd>, <Qm>
```

Unsigned result (op == 01)

```
VQMOVUN{<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

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	op		M	0	Vm			

Signed result (op == 1x)

```
VQMOVN{<c>}{<q>}.<dt> <Dd>, <Qm>
```

Unsigned result (op == 01)

```
VQMOVUN{<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

<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>	For the signed result variant: is the data type for the elements of the operand, encoded in “op<0>:size”:

op<0>	size	<dt>
0	00	S16
0	01	S32
0	10	S64
0	11	RESERVED
1	00	U16
1	01	U32
1	10	U64
1	11	RESERVED

For the unsigned result variant: is the data type for the elements of the operand, encoded in “size”:

size	<dt>
00	S16
01	S32
10	S64
11	RESERVED

<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);
        boolean sat;
        (Elem[D[d],e,esize], sat) = SatQ(operand, esize, dest_unsigned);
        if sat then FPSCR.QC = '1';
```

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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	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	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”:

size	<dt>
00	S8
01	S16
10	S32
11	RESERVED

<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
            result = -SInt(Elem[D[m+r],e,esize]);
            boolean sat;
            (Elem[D[d+r],e,esize], sat) = SignedSatQ(result, esize);
            if sat then FPSCR.QC = '1';
```

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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
(FEAT_RDM)

31	30	29	28	27	26	25	24	23	22	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			

64-bit SIMD vector (Q == 0)

```
VQRDMLAH{<q>}.<dt> <Dd>, <Dn>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VQRDMLAH{<q>}.<dt> <Qd>, <Qn>, <Qm>
```

```
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;
```

A2
(FEAT_RDM)

31	30	29	28	27	26	25	24	23	22	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				1	1	1	0	N	1	M	0	Vm			
										size																					

64-bit SIMD vector (Q == 0)

```
VQRDMLAH{<q>}.<dt> <Dd>, <Dn>, <Dm[x]>
```

128-bit SIMD vector (Q == 1)

```
VQRDMLAH{<q>}.<dt> <Qd>, <Qn>, <Dm[x]>
```

```
if !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 = TRUE;  scalar_form = TRUE;
d = UInt(D:Vd);  n = UInt(N:Vn);
regs = if Q == '0' then 1 else 2;
integer esize;
integer elements;
integer m;
integer index;
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
(FEAT_RDM)

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	1	1	N	Q	M	1	Vm			

64-bit SIMD vector (Q == 0)

```
VQRDMLAH{<q>}.<dt> <Dd>, <Dn>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VQRDMLAH{<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 = 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;
```

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
(FEAT_RDM)

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	Q	1	1	1	1	1	D	!= 11		Vn				Vd				1	1	1	0	N	1	M	0	Vm			
										size																					

64-bit SIMD vector (Q == 0)

```
VQRDMLAH{<q>}.<dt> <Dd>, <Dn>, <Dm[x]>
```

128-bit SIMD vector (Q == 1)

```
VQRDMLAH{<q>}.<dt> <Qd>, <Qn>, <Dm[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 = TRUE;  scalar_form = TRUE;
d = UInt(D:Vd);  n = UInt(N:Vn);
regs = if Q == '0' then 1 else 2;
integer esize;
integer elements;
integer m;
integer index;
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 `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”:

size	<dt>
01	S16
10	S32

<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();
integer op2;
boolean round = TRUE;
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]);
        integer rdmlah = RShr(op3 + 2*(op1*op2), esize, round);
        (result, sat) = SignedSatQ(rdmlah, 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, *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
(FEAT_RDM)

31	30	29	28	27	26	25	24	23	22	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	1	0	0	N	Q	M	1	Vm			

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 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;
```

A2
(FEAT_RDM)

31	30	29	28	27	26	25	24	23	22	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				1	1	1	1	N	1	M	0	Vm			
										size																					

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 !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 = UInt(D:Vd);  n = UInt(N:Vn);
regs = if Q == '0' then 1 else 2;
integer esize;
integer elements;
integer m;
integer index;
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
(FEAT_RDM)

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	1	0	0	N	Q	M	1	Vm			

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
(FEAT_RDM)

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	Q	1	1	1	1	1	D	!= 11		Vn				Vd				1	1	1	1	N	1	M	0	Vm			
										size																					

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 !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);
regs = if Q == '0' then 1 else 2;
integer esize;
integer m;
integer index;
integer elements;
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 `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”:

size	<dt>
01	S16
10	S32

<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();
integer op2;
boolean round = TRUE;
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]);
        integer rdmlsh = RShr(op3 - 2*(op1*op2), esize, round);
        (result, sat) = SignedSatQ(rdmlsh, esize);
        Elem[D[d+r],e,esize] = result;
        if sat then FPSCR.QC = '1';
```

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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

31	30	29	28	27	26	25	24	23	22	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	0	Vm

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

31	30	29	28	27	26	25	24	23	22	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	1	1	0	1	N	1	M	0	Vm
										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);  regs = if Q == '0' then 1 else 2;
integer esize;
integer elements;
integer m;
integer index;
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	1	1	1	1	1	0	D	size	Vn	Vd	1	0	1	1	N	Q	M	0	Vm

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;
```

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	Q	1	1	1	1	1	D	!= 11	Vn	Vd	1	1	0	1	N	1	M	0	Vm
										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);  regs = if Q == '0' then 1 else 2;
integer esize;
integer elements;
integer m;
integer index;
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 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”:

size	<dt>
01	S16
10	S32

<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.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    integer op2;
    boolean round = TRUE;
    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]);
            integer rdmulh = RShr(2*op1*op2, esize, round);
            (result, sat) = SignedSatQ(rdmulh, esize);
            Elem[D[d+r],e,esize] = result;
            if sat then FPSCR.QC = '1';
```

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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

31	30	29	28	27	26	25	24	23	22	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	0	1	N	Q	M	1	Vm

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>
```

```
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

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	0	1	N	Q	M	1	Vm

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>
```

```
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”:

U	size	<dt>
0	00	S8
0	01	S16
0	10	S32
0	11	S64
1	00	U8
1	01	U16
1	10	U32
1	11	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 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

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    bits(esize) result;
    boolean sat;
    for r = 0 to regs-1
        for e = 0 to elements-1
            integer element = Int(Elem[D[m+r], e, esize], unsigned);
            integer shift = SInt(Elem[D[n+r], e, esize]<7:0>);
            if shift >= 0 then // left shift
                element = element << shift;
            else               // rounding right shift
                shift = -shift;
                element = (element + (1 << (shift - 1))) >> shift;
            (result, sat) = SatQ(element, esize, unsigned);
            Elem[D[d+r], e, esize] = result;
            if sat then FPSCR.QC = '1';
```

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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, any CONSTRAINED UNPREDICTABLE behavior, and any operational information 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
																				op				

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	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	1	x	M	0	Vm
																				op				

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”:

op<0>	size	<dt>
0	00	S16
0	01	S32
0	10	S64
0	11	RESERVED
1	00	U16
1	01	U32
1	10	U64
1	11	RESERVED

<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.

VQRSHRN, VQRSHRUN

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) .

A1

31	30	29	28	27	26	25	24	23	22	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 (!(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 IN {'000xxx'} then SEE "Related encodings";
if U == '0' && op == '0' then SEE "VRSHRN";
if Vm<0> == '1' then UNDEFINED;
integer esize;
integer elements;
integer shift_amount;
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

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	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 IN {'000xxx'} then SEE "Related encodings";
if U == '0' && op == '0' then SEE "VRSHRN";
if Vm<0> == '1' then UNDEFINED;
integer esize;
integer elements;
integer shift_amount;
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”:

U	<type>
0	S
1	U

For the unsigned result variant: is the data type for the elements of the vectors, encoded in “U”:

U	<type>
1	S

<size> Is the data size for the elements of the vectors, encoded in “imm6<5:3>”:

imm6<5:3>	<size>
001	16
01x	32
1xx	64

<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();
    boolean round = TRUE;
    for e = 0 to elements-1
        operand = Int(Elem[Qin[m>>1],e,2*esize], src_unsigned);
        integer rshrn = RShr(operand, shift_amount, round);
        (result, sat) = SatQ(rshrn, esize, dest_unsigned);
        Elem[D[d],e,esize] = result;
        if sat then FPSCR.QC = '1';
```

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, any CONSTRAINED UNPREDICTABLE behavior, and any operational information 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	0	1	M	0	Vm			
																								op							

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			
																								op							

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”:

size	<dt>
00	S16
01	S32
10	S64
11	RESERVED

<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.

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	Vd	0	1	0	0	N	Q	M	1	Vm

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

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	0	0	N	Q	M	1	Vm

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*.

<dt> Is the data type for the elements of the vectors, encoded in “U:size”:

U	size	<dt>
0	00	S8
0	01	S16
0	10	S32
0	11	S64
1	00	U8
1	01	U16
1	10	U32
1	11	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 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

```
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);
            boolean sat;
            bits(esize) result;
            if shift >= 0 then
                (result,sat) = SatQ(operand << shift, esize, unsigned);
            else
                (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_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

VQSHL, VQSHLU (immediate)

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, *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	1	D	imm6						Vd				0	1	1	op	L	Q	M	1	Vm			

64-bit SIMD vector, signed result (!(imm6 == 000xxx && L == 0) && op == 1 && Q == 0)

```
VQSHL{<c>}{<q>}.<type><size> {<Dd>,} <Dm>, #<imm>
```

64-bit SIMD vector, unsigned result (U == 1 && !(imm6 == 000xxx && L == 0) && op == 0 && Q == 0)

```
VQSHLU{<c>}{<q>}.<type><size> {<Dd>,} <Dm>, #<imm>
```

128-bit SIMD vector, signed result (!(imm6 == 000xxx && L == 0) && op == 1 && Q == 1)

```
VQSHL{<c>}{<q>}.<type><size> {<Qd>,} <Qm>, #<imm>
```

128-bit SIMD vector, unsigned result (U == 1 && !(imm6 == 000xxx && L == 0) && op == 0 && Q == 1)

```
VQSHLU{<c>}{<q>}.<type><size> {<Qd>,} <Qm>, #<imm>
```

```
if (L:imm6) IN {'0000xxx'} then SEE "Related encodings";
if U == '0' && op == '0' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
integer esize;
integer elements;
integer shift_amount;
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;
```

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	imm6						Vd				0	1	1	op	L	Q	M	1	Vm			

64-bit SIMD vector, signed result (!(imm6 == 000xxx && L == 0) && op == 1 && Q == 0)

```
VQSHL{<c>}{<q>}.<type><size> {<Dd>,} <Dm>, #<imm>
```

64-bit SIMD vector, unsigned result (U == 1 && !(imm6 == 000xxx && L == 0) && op == 0 && Q == 0)

```
VQSHLU{<c>}{<q>}.<type><size> {<Dd>,} <Dm>, #<imm>
```

128-bit SIMD vector, signed result (!(imm6 == 000xxx && L == 0) && op == 1 && Q == 1)

```
VQSHL{<c>}{<q>}.<type><size> {<Qd>,} <Qm>, #<imm>
```

128-bit SIMD vector, unsigned result (U == 1 && !(imm6 == 000xxx && L == 0) && op == 0 && Q == 1)

```
VQSHLU{<c>}{<q>}.<type><size> {<Qd>,} <Qm>, #<imm>
```

```
if (L:imm6) IN {'0000xxx'} then SEE "Related encodings";
if U == '0' && op == '0' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
integer esize;
integer elements;
integer shift_amount;
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;
```

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”:

U	<type>
0	S
1	U

<size> Is the data size for the elements of the vectors, encoded in “L:imm6<5:3>”:

L	imm6<5:3>	<size>
0	001	8
0	01x	16
0	1xx	32
1	xxx	64

<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';
```

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 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, any CONSTRAINED UNPREDICTABLE behavior, and any operational information 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			
																								op							

Signed result

```
VQSHRN{<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	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	1	x	M	0	Vm			
																								op							

Signed result

```
VQSHRN{<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":

op<0>	size	<dt>
0	00	S16
0	01	S32
0	10	S64
0	11	RESERVED
1	00	U16
1	01	U32
1	10	U64
1	11	RESERVED

<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.

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

31	30	29	28	27	26	25	24	23	22	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	0	M	1	Vm

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>
```

```
if imm6 IN {'000xxx'} then SEE "Related encodings";
if U == '0' && op == '0' then SEE "VSHRN";
if Vm<0> == '1' then UNDEFINED;
integer esize;
integer elements;
integer shift_amount;
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

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	imm6	Vd	1	0	0	op	0	0	M	1	Vm

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>
```

```
if imm6 IN {'000xxx'} then SEE "Related encodings";
if U == '0' && op == '0' then SEE "VSHRN";
if Vm<0> == '1' then UNDEFINED;
integer esize;
integer elements;
integer shift_amount;
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”:

U	<type>
0	S
1	U

For the unsigned result variant: is the data type for the elements of the vectors, encoded in “U”:

U	<type>
1	S

<size> Is the data size for the elements of the vectors, encoded in “imm6<5:3>”:

imm6<5:3>	<size>
001	16
01x	32
1xx	64

<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();
    for e = 0 to elements-1
        operand = Int(Elem[Qin[m>>1],e,2*esize], src_unsigned);
        (result, sat) = SatQ(operand >> shift_amount, esize, dest_unsigned);
        Elem[D[d],e,esize] = result;
        if sat then FPSCR.QC = '1';
```

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, any CONSTRAINED UNPREDICTABLE behavior, and any operational information 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	0	1	M	0	Vm			
																								op							

Unsigned result

```
VQSHRUN{<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			
																								op							

Unsigned result

```
VQSHRUN{<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”:

size	<dt>
00	S16
01	S32
10	S64
11	RESERVED

<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	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

<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":

U	size	<dt>
0	00	S8
0	01	S16
0	10	S32
0	11	S64
1	00	U8
1	01	U16
1	10	U32
1	11	U64

<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
            diff = Int(Elem[D[n+r],e,esize], unsigned) - Int(Elem[D[m+r],e,esize], unsigned);
            boolean sat;
            (Elem[D[d+r],e,esize], sat) = SatQ(diff, esize, unsigned);
            if sat then FPSCR.QC = '1';
```

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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 *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	!= 11	Vn	Vd	0 1 0 0	N	0	M	0	Vm
					size								

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

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	!= 11	Vn	Vd	0 1 0 0	N	0	M	0	Vm
				size								

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”:

size	<dt>
00	I16
01	I32
10	I64

<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();
    boolean round = TRUE;
    for e = 0 to elements-1
        result = RShr(UInt(Elem[Qin[n>>1],e,2*esize] + Elem[Qin[m>>1],e,2*esize]), esize, round);
        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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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) .

A1

31	30	29	28	27	26	25	24	23	22	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	0	F	0	Q	M	0	Vm			

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');
integer esize;
integer elements;
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	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	1	Vd				0	1	0	F	0	Q	M	0	Vm			

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');
integer esize;
integer elements;
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”:

F	size	<dt>
0	10	U32
1	01	F16
1	10	F32

<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

```
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] = FPRecipEstimate(Elem[D[m+r],e,esize], StandardFPSCRValue());
            else
                Elem[D[d+r],e,esize] = UnsignedRecipEstimate(Elem[D[m+r],e,esize]);
```

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

VRECPS

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;
integer esize;
integer elements;
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	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;
if sz == '1' && InITBlock() then UNPREDICTABLE;
integer esize;
integer elements;
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”:

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.

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]);
```

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

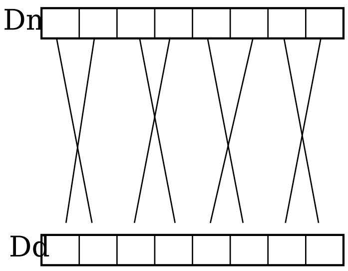
VREV16

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



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	0	1	0	Q	M	0	Vm			
																							op								

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

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		0	0	Vd				0	0	0	1	0	Q	M	0	Vm			
																							op								

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;
```

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”:

size	<dt>
00	8
01	RESERVED
1x	RESERVED

<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:

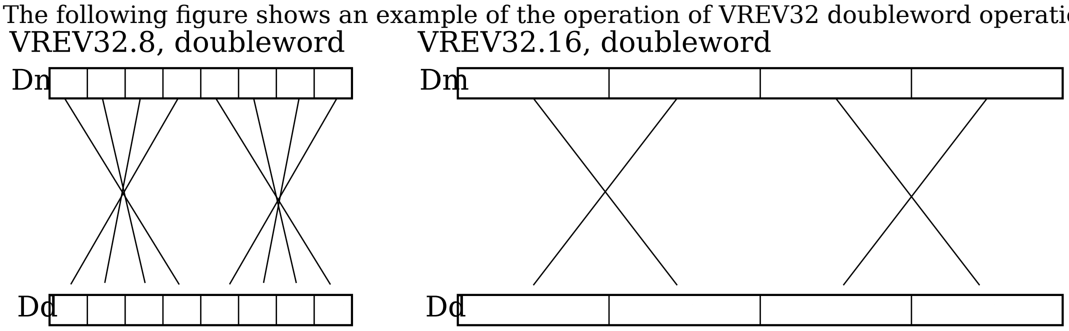
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

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 *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	0	0	1	Q	M	0	Vm			
																							op								

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 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	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	0	0	1	Q	M	0	Vm			
																							op								

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 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”:

size	<dt>
00	8
01	16
1x	RESERVED

<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:

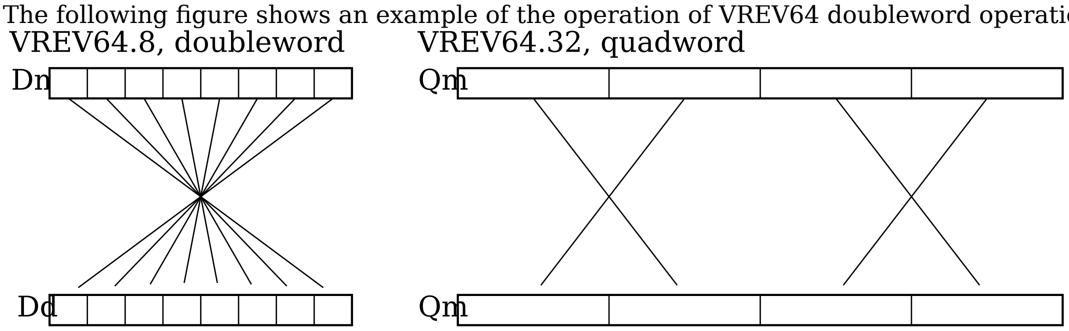
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

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.



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	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;
```

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	0	0	Vd	0	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”:

size	<dt>
00	8
01	16
10	32
11	RESERVED

<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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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

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	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”:

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.

<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 = Int(Elem[D[n+r],e,esize], unsigned);
            op2 = Int(Elem[D[m+r],e,esize], unsigned);
            result = (op1 + op2 + 1) >> 1;
            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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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	15-12	11	10	9	8	7	6	5	4	3-0
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
																			op					

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;
integer esize;
integer elements;
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	15-12	11	10	9	8	7	6	5	4	3-0
1	1	1	1	1	1	1	1	1	D	1	1	size	1	0	Vd	0	1	0	1	0	Q	M	0	Vm
																			op					

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;
integer esize;
integer elements;
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”:

size	<dt>
01	F16
10	F32

<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_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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

31	30	29	28	27	26	25	24	23	22	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	Vd				1	0	!= 00		0	1	M	0	Vm			
														RM								size									

Half-precision scalar (size == 01)
(FEAT_FP16)

```
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;
integer esize;
integer d;
integer m;
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	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	Vd				1	0	!= 00		0	1	M	0	Vm			
														RM								size									

Half-precision scalar (size == 01)
(FEAT_FP16)

```
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;
integer esize;
integer d;
integer m;
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;
integer esize;
integer elements;
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	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;
integer esize;
integer elements;
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”:

size	<dt>
01	F16
10	F32

<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_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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	1	0	1	D	1	1	1	0	1	1	Vd				1	0	!= 00		0	1	M	0	Vm			
														RM								size									

Half-precision scalar (size == 01)
(FEAT_FP16)

```
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;
integer esize;
integer d;
integer m;
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	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	1	1	Vd				1	0	!= 00		0	1	M	0	Vm			
														RM								size									

Half-precision scalar (size == 01)
(FEAT_FP16)

```
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;
integer esize;
integer d;
integer m;
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);
```

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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

31	30	29	28	27	26	25	24	23	22	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	0	0	Q	M	0	Vm
																			op					

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;
integer esize;
integer elements;
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	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	0	0	0	Q	M	0	Vm
																			op					

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;
integer esize;
integer elements;
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”:

size	<dt>
01	F16
10	F32

<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_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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	1	Vd	1	0	!= 00	0	1	M	0	Vm
														RM					size					

Half-precision scalar (size == 01)
(FEAT_FP16)

```
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;
integer esize;
integer d;
integer m;
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	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	1	Vd	1	0	!= 00	0	1	M	0	Vm
														RM					size					

Half-precision scalar (size == 01)
(FEAT_FP16)

```
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;
integer esize;
integer d;
integer m;
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);
```

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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	15-12	11	10	9	8	7	6	5	4	3-0
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
																			op					

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;
integer esize;
integer elements;
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	15-12	11	10	9	8	7	6	5	4	3-0
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
																			op					

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;
integer esize;
integer elements;
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”:

size	<dt>
01	F16
10	F32

<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_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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	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)
(FEAT_FP16)

```
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;
integer esize;
integer d;
integer m;
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	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	1	0	Vd				1	0	!= 00		0	1	M	0	Vm			
														RM								size									

Half-precision scalar (size == 01)
(FEAT_FP16)

```
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;
integer esize;
integer d;
integer m;
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);
```

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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

31-28	27	26	25	24	23	22	21	20	19	18	17	16	15-12	11	10	9-8	7	6	5	4	3-0
!= 1111	1	1	1	0	1	D	1	1	0	1	1	0	Vd	1	0	size	0	1	M	0	Vm
cond																	op				

Half-precision scalar (size == 01)
(FEAT_FP16)

```
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;
integer esize;
integer d;
integer m;
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	15-12	11	10	9-8	7	6	5	4	3-0
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)
(FEAT_FP16)

```
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;
integer esize;
integer d;
integer m;
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] = 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);
```

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

31	30	29	28	27	26	25	24	23	22	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	0	1	Q	M	0	Vm			

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;
integer esize;
integer elements;
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	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	0	0	1	Q	M	0	Vm			

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;
integer esize;
integer elements;
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”:

size	<dt>
01	F16
10	F32

<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_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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

31 30 29 28	27	26	25	24	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)
(FEAT_FP16)

```
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;
integer esize;
integer d;
integer m;
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	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

Half-precision scalar (size == 01)
(FEAT_FP16)

```
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;
exact = TRUE;
integer esize;
integer d;
integer m;
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] = 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_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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> <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_ZERO;  exact = FALSE;
integer esize;
integer elements;
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	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	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> <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_ZERO;  exact = FALSE;
integer esize;
integer elements;
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”:

size	<dt>
01	F16
10	F32

<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_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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

31 30 29 28	27	26	25	24	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	0	Vd	1	0	size	1	1	M	0	Vm
cond																	op				

Half-precision scalar (size == 01)
(FEAT_FP16)

```
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 FPRounding_ZERO else FPRoundingMode(FPSCR[]);
exact = FALSE;
integer esize;
integer d;
integer m;
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	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	1	1	M	0	Vm
																				op				

Half-precision scalar (size == 01)
(FEAT_FP16)

```
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;
integer esize;
integer d;
integer m;
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] = 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);
```

VRSHL

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

31	30	29	28	27	26	25	24	23	22	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	0	1	N	Q	M	0	Vm

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

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	0	1	N	Q	M	0	Vm

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”:

U	size	<dt>
0	00	S8
0	01	S16
0	10	S32
0	11	S64
1	00	U8
1	01	U16
1	10	U32
1	11	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 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

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    integer result;
    for r = 0 to regs-1
        for e = 0 to elements-1
            integer element = Int(Elem[D[m+r], e, esize], unsigned);
            integer shift = SInt(Elem[D[n+r], e, esize]<7:0>);
            if shift >= 0 then  // left shift
                result = element << shift;
            else                // rounding right shift
                shift = -shift;
                result = (element + (1 << (shift - 1))) >> 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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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

31	30	29	28	27	26	25	24	23	22	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	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) IN {'0000xxx'} then SEE "Related encodings";
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
integer esize;
integer elements;
integer shift_amount;
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	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	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) IN {'0000xxx'} then SEE "Related encodings";
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
integer esize;
integer elements;
integer shift_amount;
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”:

U	<type>
0	S
1	U

<size> Is the data size for the elements of the vectors, encoded in “L:imm6<5:3>”:

L	imm6<5:3>	<size>
0	001	8
0	01x	16
0	1xx	32
1	xxx	64

<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();
    boolean round = TRUE;
    for r = 0 to regs-1
        for e = 0 to elements-1
            result = RShr(Int(Elem[D[m+r],e,esize], unsigned), shift_amount, round);
            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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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, any CONSTRAINED UNPREDICTABLE behavior, and any operational information 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	0	D	1	0	Vn				Vd				0	0	0	1	N	Q	M	1	Vm			

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

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	0	Vn				Vd				0	0	0	1	N	Q	M	1	Vm			

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

<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, 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 *VSHRN*.

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

31	30	29	28	27	26	25	24	23	22	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 IN {'000xxx'} then SEE "Related encodings";
if Vm<0> == '1' then UNDEFINED;
integer esize;
integer elements;
integer shift_amount;
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	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	imm6	Vd	1	0	0	0	0	1	M	1	Vm

T1 (imm6 != 000xxx)

```
VRSHRN{<c>}{<q>}.I<size> <Dd>, <Qm>, #<imm>
```

```
if imm6 IN {'000xxx'} then SEE "Related encodings";
if Vm<0> == '1' then UNDEFINED;
integer esize;
integer elements;
integer shift_amount;
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>”:

imm6<5:3>	<size>
001	16
01x	32
1xx	64

<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();
    boolean round = TRUE;
    for e = 0 to elements-1
        result = RShr(UInt(Elem[Qin[m>>1],e,2*esize]), shift_amount, round);
        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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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, any CONSTRAINED UNPREDICTABLE behavior, and any operational information 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	0	0	M	0	Vm

A1

```
VRSHRN{<c>}{<q>}.<dt> <Dd>, <Qm>, #0
```

is equivalent to

```
VMOVN{<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	0	M	0	Vm

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”:

size	<dt>
00	I16
01	I32
10	I64
11	RESERVED

<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

31	30	29	28	27	26	25	24	23	22	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	0	F	1	Q	M	0	Vm

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');
integer esize;
integer elements;
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	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	1	Vd	0	1	0	F	1	Q	M	0	Vm

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');
integer esize;
integer elements;
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”:

F	size	<dt>
0	10	U32
1	01	F16
1	10	F32

<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

```
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_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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	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>
```

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
integer esize;
integer elements;
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	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>
```

```
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;
integer esize;
integer elements;
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”:

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.

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

```
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]);
```

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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	0	1	U	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) IN {'0000xxx'} then SEE "Related encodings";
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
integer esize;
integer elements;
integer shift_amount;
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	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	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) IN {'0000xxx'} then SEE "Related encodings";
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
integer esize;
integer elements;
integer shift_amount;
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”:

U	<type>
0	S
1	U

<size> Is the data size for the elements of the vectors, encoded in “L:imm6<5:3>”:

L	imm6<5:3>	<size>
0	001	8
0	01x	16
0	1xx	32
1	xxx	64

<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();
    boolean round = TRUE;
    for r = 0 to regs-1
        for e = 0 to elements-1
            result = RShr(Int(Elem[D[m+r],e,esize], unsigned), shift_amount, round);
            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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

VRSUBHN

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

31	30	29	28	27	26	25	24	23	22	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	!= 11	Vn	Vd	0	1	1	0	N	0	M	0	Vm
										size											

A1

```
VRSUBHN{<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	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	!= 11	Vn	Vd	0	1	1	0	N	0	M	0	Vm
										size											

T1

```
VRSUBHN{<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”:

size	<dt>
00	I16
01	I32
10	I64

<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();
    boolean round = TRUE;
    for e = 0 to elements-1
        result = RShr(UInt(Elem[Qin[n>>1],e,2*esize] - Elem[Qin[m>>1],e,2*esize]), esize, round);
        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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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.

Note

ID_ISAR6.DP indicates whether this instruction is supported.

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

A1

(FEAT_DotProd)

31	30	29	28	27	26	25	24	23	22	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	0	Vm			
																											U				

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

(FEAT_DotProd)

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	0	D	1	0	Vn				Vd				1	1	0	1	N	Q	M	0	Vm			
																											U				

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;
```

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.

Note

ID_ISAR6.DP indicates whether this instruction is supported.

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

A1
(FEAT_DotProd)

31	30	29	28	27	26	25	24	23	22	21	20	19-16	15-12	11	10	9	8	7	6	5	4	3-0
1	1	1	1	1	1	0	0	0	D	1	0	Vn	Vd	1	1	0	1	N	Q	M	0	Vm
																					U	

64-bit SIMD vector (Q == 0)

```
VSDOT{<q>}.S8 <Dd>, <Dn>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VSDOT{<q>}.S8 <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
(FEAT_DotProd)

15	14	13	12	11	10	9	8	7	6	5	4	3-0	15-12	11	10	9	8	7	6	5	4	3-0
1	1	1	1	1	1	0	0	0	D	1	0	Vn	Vd	1	1	0	1	N	Q	M	0	Vm
																					U	

64-bit SIMD vector (Q == 0)

```
VSDOT{<q>}.S8 <Dd>, <Dn>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VSDOT{<q>}.S8 <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

<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

```
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_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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	1	0	0	D	cc		Vn				Vd				1	0	!= 00		N	0	M	0	Vm			
																						size									

Equal, half-precision scalar (cc == 00 && size == 01)
(FEAT_FP16)

```
VSELEQ.F16 <Sd>, <Sn>, <Sm> // (Cannot be conditional)
```

Equal, single-precision scalar (cc == 00 && size == 10)

```
VSELEQ.F32 <Sd>, <Sn>, <Sm> // (Cannot be conditional)
```

Equal, double-precision scalar (cc == 00 && size == 11)

```
VSELEQ.F64 <Dd>, <Dn>, <Dm> // (Cannot be conditional)
```

Greater than or Equal, half-precision scalar (cc == 10 && size == 01)
(FEAT_FP16)

```
VSELGE.F16 <Sd>, <Sn>, <Sm> // (Cannot be conditional)
```

Greater than or Equal, single-precision scalar (cc == 10 && size == 10)

```
VSELGE.F32 <Sd>, <Sn>, <Sm> // (Cannot be conditional)
```

Greater than or Equal, double-precision scalar (cc == 10 && size == 11)

```
VSELGE.F64 <Dd>, <Dn>, <Dm> // (Cannot be conditional)
```

Greater than, half-precision scalar (cc == 11 && size == 01)
(FEAT_FP16)

```
VSELGT.F16 <Sd>, <Sn>, <Sm> // (Cannot be conditional)
```

Greater than, single-precision scalar (cc == 11 && size == 10)

```
VSELGT.F32 <Sd>, <Sn>, <Sm> // (Cannot be conditional)
```

Greater than, double-precision scalar (cc == 11 && size == 11)

```
VSELGT.F64 <Dd>, <Dn>, <Dm> // (Cannot be conditional)
```

Unordered, half-precision scalar (cc == 01 && size == 01)
(FEAT_FP16)

```
VSELVS.F16 <Sd>, <Sn>, <Sm> // (Cannot be conditional)
```

Unordered, single-precision scalar (cc == 01 && size == 10)

```
VSELVS.F32 <Sd>, <Sn>, <Sm> // (Cannot be conditional)
```

Unordered, double-precision scalar (cc == 01 && size == 11)

```
VSELVS.F64 <Dd>, <Dn>, <Dm> // (Cannot be conditional)
```

```
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
integer esize;
integer d;
integer n;
integer m;
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';
```

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	0	D	cc		Vn				Vd				1	0	!= 00		N	0	M	0	Vm			
																						size									

Equal, half-precision scalar (cc == 00 && size == 01)
(FEAT_FP16)

```
VSELEQ.F16 <Sd>, <Sn>, <Sm> // (Not permitted in IT block)
```

Equal, single-precision scalar (cc == 00 && size == 10)

```
VSELEQ.F32 <Sd>, <Sn>, <Sm> // (Not permitted in IT block)
```

Equal, double-precision scalar (cc == 00 && size == 11)

```
VSELEQ.F64 <Dd>, <Dn>, <Dm> // (Not permitted in IT block)
```

Greater than or Equal, half-precision scalar (cc == 10 && size == 01)
(FEAT_FP16)

```
VSELGE.F16 <Sd>, <Sn>, <Sm> // (Not permitted in IT block)
```

Greater than or Equal, single-precision scalar (cc == 10 && size == 10)

```
VSELGE.F32 <Sd>, <Sn>, <Sm> // (Not permitted in IT block)
```

Greater than or Equal, double-precision scalar (cc == 10 && size == 11)

```
VSELGE.F64 <Dd>, <Dn>, <Dm> // (Not permitted in IT block)
```

Greater than, half-precision scalar (cc == 11 && size == 01)
(FEAT_FP16)

```
VSELGT.F16 <Sd>, <Sn>, <Sm> // (Not permitted in IT block)
```

Greater than, single-precision scalar (cc == 11 && size == 10)

```
VSELGT.F32 <Sd>, <Sn>, <Sm> // (Not permitted in IT block)
```

Greater than, double-precision scalar (cc == 11 && size == 11)

```
VSELGT.F64 <Dd>, <Dn>, <Dm> // (Not permitted in IT block)
```

Unordered, half-precision scalar (cc == 01 && size == 01)
(FEAT_FP16)

```
VSELVS.F16 <Sd>, <Sn>, <Sm> // (Not permitted in IT block)
```

Unordered, single-precision scalar (cc == 01 && size == 10)

```
VSELVS.F32 <Sd>, <Sn>, <Sm> // (Not permitted in IT block)
```

Unordered, double-precision scalar (cc == 01 && size == 11)

```
VSELVS.F64 <Dd>, <Dn>, <Dm> // (Not permitted in IT block)
```

```
if InITBlock() then UNPREDICTABLE;
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
integer esize;
integer d;
integer n;
integer m;
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

<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(); 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];
```

VSHL (immediate)

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	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) IN {'0000xxx'} then SEE "Related encodings";
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
integer esize;
integer elements;
integer shift_amount;
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);
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	0	1	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) IN {'0000xxx'} then SEE "Related encodings";
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
integer esize;
integer elements;
integer shift_amount;
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);
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*.

<size> Is the data size for the elements of the vectors, encoded in “L:imm6<5:3>”:

L	imm6<5:3>	<size>
0	001	8
0	01x	16
0	1xx	32
1	xxx	64

<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
            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_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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

31	30	29	28	27	26	25	24	23	22	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	0	0	N	Q	M	0	Vm			

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

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	0	0	N	Q	M	0	Vm			

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”:

U	size	<dt>
0	00	S8
0	01	S16
0	10	S32
0	11	S64
1	00	U8
1	01	U16
1	10	U32
1	11	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 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

```
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>);
            integer result;
            if shift >= 0 then
                result = Int(Elem[D[m+r],e,esize], unsigned) << shift;
            else
                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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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

31	30	29	28	27	26	25	24	23	22	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	1	0	0	0	M	1	Vm			

A1 (imm6 != 000xxx)

```
VSHLL{<c>}{<q>}.<type><size> <Qd>, <Dm>, #<imm>
```

```
if imm6 IN {'000xxx'} then SEE "Related encodings";
if Vd<0> == '1' then UNDEFINED;
integer esize;
integer elements;
integer shift_amount;
case imm6 of
    when '001xxx'  esize = 8;  elements = 8;  shift_amount = UInt(imm6) - 8;
    when '01xxxx'  esize = 16;  elements = 4;  shift_amount = UInt(imm6) - 16;
    when '1xxxxx'  esize = 32;  elements = 2;  shift_amount = UInt(imm6) - 32;
if shift_amount == 0 then SEE "VMOVL";
unsigned = (U == '1');  d = UInt(D:Vd);  m = UInt(M:Vm);
```

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	1	D	1	1	size		1	0	Vd				0	0	1	1	0	0	M	0	Vm			

A2

```
VSHLL{<c>}{<q>}.<type><size> <Qd>, <Dm>, #<imm>
```

```
if size == '11' || Vd<0> == '1' then UNDEFINED;
esize = 8 << UInt(size);  elements = 64 DIV esize;  shift_amount = esize;
unsigned = FALSE;  // Or TRUE without change of functionality
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	U	1	1	1	1	1	D	imm6						Vd				1	0	1	0	0	0	M	1	Vm			

T1 (imm6 != 000xxx)

```
VSHLL{<c>}{<q>}.<type><size> <Qd>, <Dm>, #<imm>
```

```
if imm6 IN {'000xxx'} then SEE "Related encodings";
if Vd<0> == '1' then UNDEFINED;
integer esize;
integer elements;
integer shift_amount;
case imm6 of
    when '001xxx'  esize = 8;  elements = 8;  shift_amount = UInt(imm6) - 8;
    when '01xxxx'  esize = 16;  elements = 4;  shift_amount = UInt(imm6) - 16;
    when '1xxxxx'  esize = 32;  elements = 2;  shift_amount = UInt(imm6) - 32;
if shift_amount == 0 then SEE "VMOVL";
unsigned = (U == '1');  d = UInt(D:Vd);  m = UInt(M:Vm);
```

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	1	D	1	1	size		1	0	Vd				0	0	1	1	0	0	M	0	Vm			

T2

```
VSHLL{<c>}{<q>}.<type><size> <Qd>, <Dm>, #<imm>
```

```
if size == '11' || Vd<0> == '1' then UNDEFINED;
esize = 8 << UInt(size);  elements = 64 DIV esize;  shift_amount = esize;
unsigned = FALSE;  // Or TRUE without change of functionality
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 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*.

<type> The data type for the elements of the operand. It must be one of:

S
: Signed. In encoding T1/A1, encoded as U = 0.

U
: Unsigned. In encoding T1/A1, encoded as U = 1.

I
: Untyped integer, Available only in encoding T2/A2.

<size> The data size for the elements of the operand. The following table shows the permitted values and their encodings:

<size>	Encoding T1/A1	Encoding T2/A2
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

<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.

<imm> The immediate value. <imm> must lie in the range 1 to <size>, and:

- If <size> == <imm>, the encoding is T2/A2.
- Otherwise, the encoding is T1/A1, and:
    - If <size> == 8, <imm> is encoded in imm6<2:0>.
    - If <size> == 16, <imm> is encoded in imm6<3:0>.
    - If <size> == 32, <imm> 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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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) IN {'0000xxx'} then SEE "Related encodings";
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
integer esize;
integer elements;
integer shift_amount;
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	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	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) IN {'0000xxx'} then SEE "Related encodings";
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
integer esize;
integer elements;
integer shift_amount;
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”:

U	<type>
0	S
1	U

<size> Is the data size for the elements of the vectors, encoded in “L:imm6<5:3>”:

L	imm6<5:3>	<size>
0	001	8
0	01x	16
0	1xx	32
1	xxx	64

<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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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, any CONSTRAINED UNPREDICTABLE behavior, and any operational information 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	0	D	1	0	Vn				Vd				0	0	0	1	N	Q	M	1	Vm			

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

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	0	Vn				Vd				0	0	0	1	N	Q	M	1	Vm			

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

<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, 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_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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

31	30	29	28	27	26	25	24	23	22	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	0	M	1	Vm

A1 (imm6 != 000xxx)

```
VSHRN{<c>}{<q>}.I<size> <Dd>, <Qm>, #<imm>
```

```
if imm6 IN {'000xxx'} then SEE "Related encodings";
if Vm<0> == '1' then UNDEFINED;
integer esize;
integer elements;
integer shift_amount;
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	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	imm6	Vd	1	0	0	0	0	0	M	1	Vm

T1 (imm6 != 000xxx)

```
VSHRN{<c>}{<q>}.I<size> <Dd>, <Qm>, #<imm>
```

```
if imm6 IN {'000xxx'} then SEE "Related encodings";
if Vm<0> == '1' then UNDEFINED;
integer esize;
integer elements;
integer shift_amount;
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>”:

imm6<5:3>	<size>
001	16
01x	32
1xx	64

<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();
    for e = 0 to elements-1
        result = LSR(Elem[Qin[m>>1],e,2*esize], shift_amount);
        Elem[D[d],e,esize] = result<esize-1:0>;
```

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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, any CONSTRAINED UNPREDICTABLE behavior, and any operational information 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	0	0	M	0	Vm

A1

```
VSHRN{<c>}{<q>}.<dt> <Dd>, <Qm>, #0
```

is equivalent to

```
VMOVN{<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	0	M	0	Vm

T1

```
VSHRN{<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”:

size	<dt>
00	I16
01	I32
10	I64
11	RESERVED

<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.

VSLI

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

31	30	29	28	27	26	25	24	23	22	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	1	L	Q	M	1	Vm			

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> {<Qd>,} <Qm>, #<imm>
```

```
if (L:imm6) IN {'0000xxx'} then SEE "Related encodings";
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
integer esize;
integer elements;
integer shift_amount;
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);
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	imm6						Vd				0	1	0	1	L	Q	M	1	Vm			

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> {<Qd>,} <Qm>, #<imm>
```

```
if (L:imm6) IN {'0000xxx'} then SEE "Related encodings";
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
integer esize;
integer elements;
integer shift_amount;
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);
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*.

<size> Is the data size for the elements of the vectors, encoded in "L:imm6<5:3>":

L	imm6<5:3>	<size>
0	001	8
0	01x	16
0	1xx	32
1	xxx	64

<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();
    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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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
(FEAT_AA32I8MM)

31	30	29	28	27	26	25	24	23	22	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	0	Vn				Vd				1	1	0	0	N	1	M	0	Vm			
								B																			U				

A1

```
VSMMLA{<q>}.S8 <Qd>, <Qn>, <Qm>
```

```
if !HaveAArch32Int8MatMulExt() then UNDEFINED;
boolean op1_unsigned;
boolean op2_unsigned;
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
(FEAT_AA32I8MM)

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	0	Vn				Vd				1	1	0	0	N	1	M	0	Vm			
								B																			U				

T1

```
VSMMLA{<q>}.S8 <Qd>, <Qn>, <Qm>
```

```
if InITBlock() then UNPREDICTABLE;
if !HaveAArch32Int8MatMulExt() then UNDEFINED;
boolean op1_unsigned;
boolean op2_unsigned;
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.
<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) addend   = Q[d>>1];

Q[d>>1] = MatMulAdd(addend, operand1, operand2, op1_unsigned, op2_unsigned);
```

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 *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	0	1	Vd				1	0	size		1	1	M	0	Vm			
cond																															

Half-precision scalar (size == 01)
(FEAT_FP16)

```
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' && cond != '1110' then UNPREDICTABLE;
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
integer esize;
integer d;
integer m;
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	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	0	0	1	Vd				1	0	size		1	1	M	0	Vm			

Half-precision scalar (size == 01)
(FEAT_FP16)

```
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;
integer esize;
integer d;
integer m;
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

<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] = Zeros(16) : FPSqrt(S[m]<15:0>, FPSCR[]);
        when 32 S[d] = FPSqrt(S[m], FPSCR[]);
        when 64 D[d] = FPSqrt(D[m], FPSCR[]);
```

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

VSRA

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

31	30	29	28	27	26	25	24	23	22	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	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>
```

```
if (L:imm6) IN {'0000xxx'} then SEE "Related encodings";
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
integer esize;
integer elements;
integer shift_amount;
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	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	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>
```

```
if (L:imm6) IN {'0000xxx'} then SEE "Related encodings";
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
integer esize;
integer elements;
integer shift_amount;
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”:

U	<type>
0	S
1	U

<size> Is the data size for the elements of the vectors, encoded in “L:imm6<5:3>”:

L	imm6<5:3>	<size>
0	001	8
0	01x	16
0	1xx	32
1	xxx	64

<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] = 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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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) IN {'0000xxx'} then SEE "Related encodings";
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
integer esize;
integer elements;
integer shift_amount;
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	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	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) IN {'0000xxx'} then SEE "Related encodings";
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
integer esize;
integer elements;
integer shift_amount;
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

<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 "L:imm6<5:3>":

L	imm6<5:3>	<size>
0	001	8
0	01x	16
0	1xx	32
1	xxx	64

<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();
    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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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	19-16	15-12	11	10	9	8	7-6	5-4	3-0
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>
```

```
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 then UNPREDICTABLE;
```

A2

31	30	29	28	27	26	25	24	23	22	21	20	19-16	15-12	11	10	9	8	7-6	5-4	3-0
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;
```

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.

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	D	0	0	Rn				Vd				0	1	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 = 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`.
- 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.

A4

31	30	29	28	27	26	25	24	23	22	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	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;
```

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

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	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>
```

```
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 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	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;
```

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

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	0	0	Rn				Vd				0	1	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 = 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`.
- 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	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	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;
```

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.

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”:

size	<size>
00	8
01	16
10	32
11	64

<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*.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();

    address = R[n];

    boolean nontemporal = FALSE;
    boolean tagchecked  = FALSE;
    AccessDescriptor accdesc = CreateAccDescASIMD(MemOp_STORE, nontemporal, tagchecked);
    if !IsAligned(address, alignment) then
        AArch32.Abort(address, AlignmentFault(accdesc));

    for r = 0 to regs-1
        for e = 0 to elements-1
            if ebytes != 8 then
                MemU[address,ebytes] = Elem[D[d+r],e,8*ebytes];
            else
                if !IsAligned(address, ebytes) && AlignmentEnforced() then
                    AArch32.Abort(address, AlignmentFault(accdesc));

                bits(64) data = Elem[D[d+r],e,64];
                if BigEndian(AccessType_ASIMD) then
                    MemU[address,4] = data<63:32>;
                    MemU[address+4,4] = data<31:0>;
                else
                    MemU[address,4] = data<31:0>;
                    MemU[address+4,4] = 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;
```

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_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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

31	30	29	28	27	26	25	24	23	22	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	0	index_align	Rm
														size					

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

31	30	29	28	27	26	25	24	23	22	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	0	0	index_align	Rm
														size					

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;
```

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	0	0	Rn	Vd	1	0	0	0	index_align	Rm
														size					

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

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	0	0	Rn	Vd	0	0	0	0	index_align	Rm
														size					

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;
```

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	0	0	Rn	Vd	0	1	0	0	index_align	Rm
														size					

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

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	0	0	Rn	Vd	1 0	0 0	index_align	Rm
														size			

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”:

size	<size>
00	8
01	16
10	32

<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 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];

    boolean nontemporal = FALSE;
    boolean tagchecked  = FALSE;
    AccessDescriptor accdesc = CreateAccDescASIMD(MemOp_STORE, nontemporal, tagchecked);
    if !IsAligned(address, alignment) then
        AArch32.Abort(address, AlignmentFault(accdesc));

    MemU[address,ebytes] = Elem[D[d],index,8*ebytes];
    if wback then
        if register_index then
            R[n] = R[n] + R[m];
        else
            R[n] = R[n] + ebytes;
```

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_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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	19-16	15-12	11	10	9	8	7-6	5-4	3-0
1	1	1	1	0	1	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>
```

```
pairs = 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+pairs > 32 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `d2+pairs > 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	19-16	15-12	11	10	9	8	7-6	5-4	3-0
1	1	1	1	0	1	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 != 11x1)

```
VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

```
pairs = 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+pairs > 32 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `d2+pairs > 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

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

```
pairs = 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+pairs > 32 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `d2+pairs > 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

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	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 != 11x1)

```
VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

```
pairs = 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+pairs > 32 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `d2+pairs > 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 *VST2 (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”:

size	<size>
00	8
01	16
10	32
11	RESERVED

<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];

    boolean nontemporal = FALSE;
    boolean tagchecked  = FALSE;
    AccessDescriptor accdesc = CreateAccDescASIMD(MemOp_STORE, nontemporal, tagchecked);
    if !IsAligned(address, alignment) then
        AArch32.Abort(address, AlignmentFault(accdesc));

    for r = 0 to pairs-1
        for e = 0 to elements-1
            MemU[address,       ebytes] = Elem[D[d+r], e,8*ebytes];
            MemU[address+ebytes,ebytes] = Elem[D[d2+r],e,8*ebytes];
            address = address + 2*ebytes;
    if wback then
        if register_index then
            R[n] = R[n] + R[m];
        else
            R[n] = R[n] + 16*pairs;
```

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_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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
														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;
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.

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	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 != 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

31	30	29	28	27	26	25	24	23	22	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				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 != 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

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	0	0	Rn				Vd				0	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 != 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

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	0	0	Rn				Vd				0	1	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 != 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.

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	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 != 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.

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

<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 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];

    boolean nontemporal = FALSE;
    boolean tagchecked  = FALSE;
    AccessDescriptor accdesc = CreateAccDescASIMD(MemOp_STORE, nontemporal, tagchecked);
    if !IsAligned(address, alignment) then
        AArch32.Abort(address, AlignmentFault(accdesc));

    MemU[address,       ebytes] = Elem[D[d], index,8*ebytes];
    MemU[address+ebytes,ebytes] = Elem[D[d2],index,8*ebytes];
    if wback then
        if register_index then
            R[n] = R[n] + R[m];
        else
            R[n] = R[n] + 2*ebytes;
```

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_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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	1	0	1	0	0	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;
integer inc;
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.

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	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;
integer inc;
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 CONSTRAINED 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

<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”:

size	<size>
00	8
01	16
10	32
11	RESERVED

<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> }
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.

<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 the variants of this instruction, see *Advanced SIMD addressing mode*.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();

    address = R[n];

    boolean nontemporal = FALSE;
    boolean tagchecked  = FALSE;
    AccessDescriptor accdesc = CreateAccDescASIMD(MemOp_STORE, nontemporal, tagchecked);
    if !IsAligned(address, alignment) then
        AArch32.Abort(address, AlignmentFault(accdesc));

    for e = 0 to elements-1
        MemU[address,          ebytes] = Elem[D[d], e,8*ebytes];
        MemU[address+ebytes,   ebytes] = Elem[D[d2],e,8*ebytes];
        MemU[address+2*ebytes,ebytes] = Elem[D[d3],e,8*ebytes];
        address = address + 3*ebytes;
    if wback then
        if register_index then
            R[n] = R[n] + R[m];
        else
            R[n] = R[n] + 24;
```

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_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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	1	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

31	30	29	28	27	26	25	24	23	22	21	20	19-16	15-12	11	10	9	8	7-4	3-0
1	1	1	1	0	1	0	0	1	D	0	0	Rn	Vd	1	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<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.

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	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.

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	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.

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	0	0	Rn				Vd				1	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<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”:

size	<size>
00	8
01	16
10	32

<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,8*ebytes];
    MemU[address+ebytes,  ebytes] = Elem[D[d2],index,8*ebytes];
    MemU[address+2*ebytes,ebytes] = Elem[D[d3],index,8*ebytes];
    if wback then
        if register_index then
            R[n] = R[n] + R[m];
        else
            R[n] = R[n] + 3*ebytes;
```

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_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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;
integer inc;
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.

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	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;
integer inc;
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”:

size	<size>
00	8
01	16
10	32
11	RESERVED

<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];

    boolean nontemporal = FALSE;
    boolean tagchecked  = FALSE;
    AccessDescriptor accdesc = CreateAccDescASIMD(MemOp_STORE, nontemporal, tagchecked);
    if !IsAligned(address, alignment) then
        AArch32.Abort(address, AlignmentFault(accdesc));

    for e = 0 to elements-1
        MemU[address,         ebytes] = Elem[D[d], e,8*ebytes];
        MemU[address+ebytes,  ebytes] = Elem[D[d2],e,8*ebytes];
        MemU[address+2*ebytes,ebytes] = Elem[D[d3],e,8*ebytes];
        MemU[address+3*ebytes,ebytes] = Elem[D[d4],e,8*ebytes];
        address = address + 4*ebytes;
    if wback then
        if register_index then
            R[n] = R[n] + R[m];
        else
            R[n] = R[n] + 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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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	19-16	15-12	11	10	9	8	7-4	3-0
1	1	1	1	0	1	0	0	1	D	0	0	Rn	Vd	0	0	1	1	index_align	Rm
														size					

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 = 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`.
- 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	19-16	15-12	11	10	9	8	7-4	3-0
1	1	1	1	0	1	0	0	1	D	0	0	Rn	Vd	0	1	1	1	index_align	Rm
														size					

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.

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	0	0	Rn	Vd	1 0	1 1	index_align	Rm
														size			

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 != '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.

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	D	0	0	Rn				Vd				0	0	1	1	index_align				Rm			
																				size											

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 = 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.
- 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

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	0	0	Rn				Vd				0	1	1	1	index_align				Rm			
																				size											

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

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	0	0	Rn	Vd	1 0	1 1	index_align	Rm
														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.

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”:

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:

{ <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];

    boolean nontemporal = FALSE;
    boolean tagchecked  = FALSE;
    AccessDescriptor accdesc = CreateAccDescASIMD(MemOp_STORE, nontemporal, tagchecked);
    if !IsAligned(address, alignment) then
        AArch32.Abort(address, AlignmentFault(accdesc));

    MemU[address,         ebytes] = Elem[D[d], index,8*ebytes];
    MemU[address+ebytes,  ebytes] = Elem[D[d2],index,8*ebytes];
    MemU[address+2*ebytes,ebytes] = Elem[D[d3],index,8*ebytes];
    MemU[address+3*ebytes,ebytes] = Elem[D[d4],index,8*ebytes];
    if wback then
        if register_index then
            R[n] = R[n] + R[m];
        else
            R[n] = R[n] + 4*ebytes;
```

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_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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 *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*.

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
cond												imm8<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 == '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) > 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
cond											

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

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	P	U	D	W	0	Rn	Vd	1 0	1 1	imm8<7:1>	0
											imm8<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 == '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) > 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

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	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.

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*.

<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

Alias	Is preferred when
VPUSH	`P == '1' && U == '0' && W == '1' && Rn == '1101'`

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
            MemA[address,4] = S[d+r];
            address = address+4;
        else
            // Store as two word-aligned words in the correct order for current endianness.
            if BigEndian(AccessType_ASIMD) then
                MemA[address,4] =  D[d+r]<63:32>;
                MemA[address+4,4] =  D[d+r]<31:0>;
            else
                MemA[address,4] = D[d+r]<31:0>;
                MemA[address+4,4] = D[d+r]<63:32>;

            address = address+8;

    if wback then R[n] = if add then R[n]+imm32 else R[n]-imm32;
```

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_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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

31 30 29 28	27 26 25	24	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	1	U	D	0	0	Rn	Vd	1 0	size	imm8
cond											

Half-precision scalar (size == 01)
(FEAT_FP16)

```
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' && 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);
integer d;
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

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	1	U	D	0	0	Rn	Vd	1 0	size	imm8

Half-precision scalar (size == 01)
(FEAT_FP16)

```
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);
integer d;
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' && 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*.

.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”:

U	+/-
0	-
1	+

<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.

Operation

```
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] = S[d]<15:0>;
        when 32
            MemA[address,4] = S[d];
        when 64
            // Store as two word-aligned words in the correct order for current endianness.
            if BigEndian(AccessType_ASIMD) then
                MemA[address,4]   = D[d]<63:32>;
                MemA[address+4,4] = D[d]<31:0>;
            else
                MemA[address,4]   = D[d]<31:0>;
                MemA[address+4,4] = D[d]<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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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;
integer esize;
integer elements;
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)
(FEAT_FP16)

```
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;
integer esize;
integer d;
integer n;
integer m;
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	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	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;
if sz == '1' && InITBlock() then UNPREDICTABLE;
advsimd = TRUE;
integer esize;
integer elements;
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	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	D	1	1	Vn				Vd				1	0	size		N	1	M	0	Vm			

Half-precision scalar (size == 01)
(FEAT_FP16)

```
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;
integer esize;
integer d;
integer n;
integer m;
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”:

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
                Elem[D[d+r],e,esize] = FPSub(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) : 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[]);
```

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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	0	D	size		Vn				Vd				1	0	0	0	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;
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	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)

```
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;
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 vectors, encoded in “size”:

size	<dt>
00	I8
01	I16
10	I32
11	I64

<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
            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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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	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”:

size	<dt>
00	I16
01	I32
10	I64

<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 = 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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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

31	30	29	28	27	26	25	24	23	22	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	0	N	0	M	0	Vm
										size						op					

A1

```
VSUBL{<c>}{<q>}.<dt> <Qd>, <Dn>, <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_vsubw = (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	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	1	0	N	0	M	0	Vm
										size						op					

T1

```
VSUBL{<c>}{<q>}.<dt> <Qd>, <Dn>, <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_vsubw = (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”:

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

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for e = 0 to elements-1
        integer op1;
        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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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					

A1

```
VSUBW{<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_vsubw = (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	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	1	1	N	0	M	0	Vm
										size						op					

T1

```
VSUBW{<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_vsubw = (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”:

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.

<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

```
if ConditionPassed() then
    EncodingSpecificOperations();  CheckAdvSIMDEnabled();
    for e = 0 to elements-1
        integer op1;
        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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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

(FEAT_AA32I8MM)

31	30	29	28	27	26	25	24	23	22	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	0	0	Vn				Vd				1	1	0	1	N	Q	M	1	Vm			
																											U				

64-bit SIMD vector (Q == 0)

```
VSUDOT{<q>}.U8 <Dd>, <Dn>, <Dm>[<index>]
```

128-bit SIMD vector (Q == 1)

```
VSUDOT{<q>}.U8 <Qd>, <Qn>, <Dm>[<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

(FEAT_AA32I8MM)

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	1	D	0	0	Vn				Vd				1	1	0	1	N	Q	M	1	Vm			
																											U				

64-bit SIMD vector (Q == 0)

```
VSUDOT{<q>}.U8 <Dd>, <Dn>, <Dm>[<index>]
```

128-bit SIMD vector (Q == 1)

```
VSUDOT{<q>}.U8 <Qd>, <Qn>, <Dm>[<index>]
```

```
if InITBlock() then UNPREDICTABLE;
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

```
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_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

VSWP

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	15-12	11	10	9	8	7	6	5	4	3-0
1	1	1	1	0	0	1	1	1	D	1	1	0	0	1	0	Vd	0	0	0	0	0	Q	M	0	Vm
												size													

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	15-12	11	10	9	8	7	6	5	4	3-0
1	1	1	1	1	1	1	1	1	D	1	1	0	0	1	0	Vd	0	0	0	0	0	Q	M	0	Vm
												size													

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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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

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

T1

15	14	13	12	11	10	9	8	7	6	5	4	3-0	15-12	11	10	9-8	7	6	5	4	3-0
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.

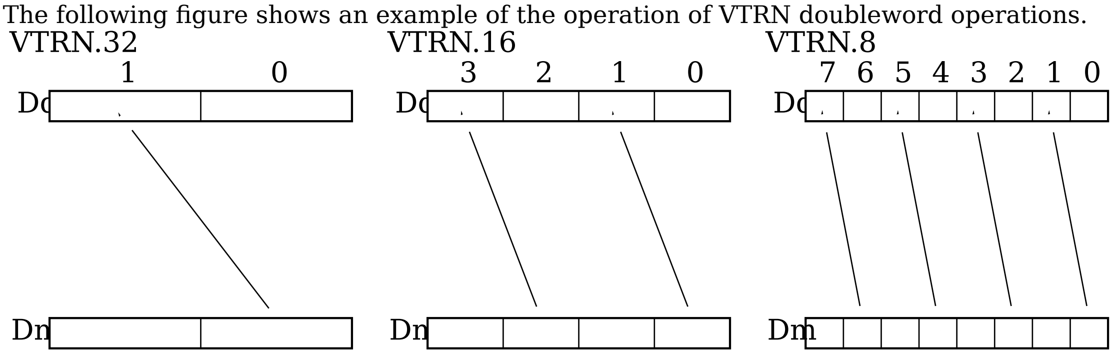
Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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.



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

31	30	29	28	27	26	25	24	23	22	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	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

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	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”:

size	<dt>
00	8
01	16
10	32
11	RESERVED

<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_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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

31	30	29	28	27	26	25	24	23	22	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	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> {<Qd>,} <Qn>, <Qm>
```

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size == '11' 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

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> {<Qd>,} <Qn>, <Qm>
```

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size == '11' 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.
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”:

size	<dt>
00	8
01	16
10	32

<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
            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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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 OPTIONAL instruction. From Armv8.4 it is mandatory for all implementations to support it.

Note

ID_ISAR6.DP indicates whether this instruction is supported.

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

A1
(FEAT_DotProd)

31	30	29	28	27	26	25	24	23	22	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			
																											U				

64-bit SIMD vector (Q == 0)

```
VUDOT{<q>}.U8 <Dd>, <Dn>, <Dm>[<index>]
```

128-bit SIMD vector (Q == 1)

```
VUDOT{<q>}.U8 <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
(FEAT_DotProd)

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	0	D	1	0	Vn				Vd				1	1	0	1	N	Q	M	1	Vm			
																											U				

64-bit SIMD vector (Q == 0)

```
VUDOT{<q>}.U8 <Dd>, <Dn>, <Dm>[<index>]
```

128-bit SIMD vector (Q == 1)

```
VUDOT{<q>}.U8 <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;
```

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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.

Note

ID_ISAR6.DP indicates whether this instruction is supported.

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

A1
(FEAT_DotProd)

31	30	29	28	27	26	25	24	23	22	21	20	19-16	15-12	11	10	9	8	7	6	5	4	3-0
1	1	1	1	1	1	0	0	0	D	1	0	Vn	Vd	1	1	0	1	N	Q	M	1	Vm
																					U	

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
(FEAT_DotProd)

15	14	13	12	11	10	9	8	7	6	5	4	3-0	15-12	11	10	9	8	7	6	5	4	3-0
1	1	1	1	1	1	0	0	0	D	1	0	Vn	Vd	1	1	0	1	N	Q	M	1	Vm
																					U	

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

<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

```
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_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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
(FEAT_AA32I8MM)

31	30	29	28	27	26	25	24	23	22	21	20	19-16	15-12	11	10	9	8	7	6	5	4	3-0
1	1	1	1	1	1	0	0	0	D	1	0	Vn	Vd	1	1	0	0	N	1	M	1	Vm
								B													U	

A1

```
VUMMLA{<q>}.U8 <Qd>, <Qn>, <Qm>
```

```
if !HaveAArch32Int8MatMulExt() then UNDEFINED;
boolean op1_unsigned;
boolean op2_unsigned;
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
(FEAT_AA32I8MM)

15	14	13	12	11	10	9	8	7	6	5	4	3-0	15-12	11	10	9	8	7	6	5	4	3-0
1	1	1	1	1	1	0	0	0	D	1	0	Vn	Vd	1	1	0	0	N	1	M	1	Vm
								B													U	

T1

```
VUMMLA{<q>}.U8 <Qd>, <Qn>, <Qm>
```

```
if InITBlock() then UNPREDICTABLE;
if !HaveAArch32Int8MatMulExt() then UNDEFINED;
boolean op1_unsigned;
boolean op2_unsigned;
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.

<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) addend   = Q[d>>1];

Q[d>>1] = MatMulAdd(addend, operand1, operand2, op1_unsigned, op2_unsigned);
```

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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

(FEAT_AA32I8MM)

31	30	29	28	27	26	25	24	23	22	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	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 <Qd>, <Qn>, <Dm>[<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

(FEAT_AA32I8MM)

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	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 <Qd>, <Qn>, <Dm>[<index>]
```

```
if InITBlock() then UNPREDICTABLE;
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

```
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_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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
(FEAT_AA32I8MM)

31	30	29	28	27	26	25	24	23	22	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	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>
```

```
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
(FEAT_AA32I8MM)

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	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>
```

```
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 second SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
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_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

VUSMMLA

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 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
(FEAT_AA32I8MM)

31	30	29	28	27	26	25	24	23	22	21	20	19-16	15-12	11	10	9	8	7	6	5	4	3-0
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	

A1

```
VUSMMLA{<q>}.S8 <Qd>, <Qn>, <Qm>
```

```
if !HaveAArch32Int8MatMulExt() then UNDEFINED;
boolean op1_unsigned;
boolean op2_unsigned;
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
(FEAT_AA32I8MM)

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

T1

```
VUSMMLA{<q>}.S8 <Qd>, <Qn>, <Qm>
```

```
if InITBlock() then UNPREDICTABLE;
if !HaveAArch32Int8MatMulExt() then UNDEFINED;
boolean op1_unsigned;
boolean op2_unsigned;
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.

<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) addend   = Q[d>>1];

Q[d>>1] = MatMulAdd(addend, operand1, operand2, op1_unsigned, op2_unsigned);
```

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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.8, doubleword

	Register state before opera								Register state after operation							
Dd	A7	A6	A5	A4	A3	A2	A1	A0	B6	B4	B2	B0	A6	A4	A2	A0
Dm	B7	B6	B5	B4	B3	B2	B1	B0	B7	B5	B3	B1	A7	A5	A3	A1

The following figure shows an example of the operation of VUZP quadword operation for data type 32.

VUZP.32, quadword

	Register state before opera				Register state after operation			
Qd	A3	A2	A1	A0	B2	B0	A2	A0
Qm	B3	B2	B1	B0	B3	B1	A3	A1

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;
quadword_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	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;
quadword_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*.

<q> See *Standard assembler syntax fields*.

<dt> For the 64-bit SIMD vector variant: is the data type for the elements of the vectors, encoded in “size”:

size	<dt>
00	8
01	16
1x	RESERVED

For the 128-bit SIMD vector variant: is the data type for the elements of the vectors, encoded in “size”:

size	<dt>
00	8
01	16
10	32
11	RESERVED

<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();
    if quadword_operation then
        if d == m then
            Q[d>>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;
        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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

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, any CONSTRAINED UNPREDICTABLE behavior, and any operational information 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	0	0	1	0	M	0	Vm
																					Q			

64-bit SIMD vector

```
VUZP{<c>}{<q>}.32 <Dd>, <Dm>
```

is equivalent to

```
VTRN{<c>}{<q>}.32 <Dd>, <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	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

```
VUZP{<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.

VZIP

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

	Register state before opera								Register state after operation							
Dd	A7	A6	A5	A4	A3	A2	A1	A0	B3	A3	B2	A2	B1	A1	B0	A0
Dm	B7	B6	B5	B4	B3	B2	B1	B0	B7	A7	B6	A6	B5	A5	B4	A4

The following figure shows an example of the operation of VZIP quadword operation for data type 32.

VZIP.32, quadword

	Register state before opera				Register state after operation			
Qd	A3	A2	A1	A0	B1	A1	B0	A0
Qm	B3	B2	B1	B0	B3	A3	B2	A2

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	1	Q	M	0	Vm			

64-bit SIMD vector (Q == 0)

```
VZIP{<c>}{<q>}.<dt> <Dd>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VZIP{<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;
quadword_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	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	1	Q	M	0	Vm			

64-bit SIMD vector (Q == 0)

```
VZIP{<c>}{<q>}.<dt> <Dd>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VZIP{<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;
quadword_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*.

<q> See *Standard assembler syntax fields*.

<dt> For the 64-bit SIMD vector variant: is the data type for the elements of the vectors, encoded in “size”:

size	<dt>
00	8
01	16
1x	RESERVED

For the 128-bit SIMD vector variant: is the data type for the elements of the vectors, encoded in “size”:

size	<dt>
00	8
01	16
10	32
11	RESERVED

<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();
    if quadword_operation then
        if d == m then
            Q[d>>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;
        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, any CONSTRAINED UNPREDICTABLE behavior, and any operational information 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	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

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	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.

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19

Top-level encodings for A32

31–28	27–25	24–5	4	3–0
cond	op0		op1	

Decode fields cond	op0	op1	Instruction details
!= 1111	00x		Data-processing and miscellaneous instructions
!= 1111	010		Load/Store Word, Unsigned Byte (immediate, literal)
!= 1111	011	0	Load/Store Word, Unsigned Byte (register)
!= 1111	011	1	Media instructions
	10x		Branch, branch with link, and block data transfer
	11x		System register access, Advanced SIMD, floating-point, and Supervisor call
1111	0xx		Unconditional instructions

Data-processing and miscellaneous instructions

These instructions are under the top-level.

31–28	27–26	25	24–20	19–8	7	6–5	4	3–0
!= 1111	00	op0	op1		op2	op3	op4	

Decode fields op0	op1	op2	op3	op4	Instruction details
0		1	!= 00	1	Extra load/store
0	0xxxx	1	00	1	Multiply and Accumulate
0	1xxxx	1	00	1	Synchronization primitives and Load-Acquire/Store-Release
0	10xx0	0			Miscellaneous
0	10xx0	1		0	Halfword Multiply and Accumulate
0	!= 10xx0			0	Data-processing register (immediate shift)
0	!= 10xx0	0		1	Data-processing register (register shift)
1					Data-processing immediate

Extra load/store

These instructions are under Data-processing and miscellaneous instructions.

31–28	27–25	24–23	22	21–8	7	6–5	4	3–0
!= 1111	000		op0		1	!= 00	1	

Decode fields op0	Instruction details
0	Load/Store Dual, Half, Signed Byte (register)
1	Load/Store Dual, Half, Signed Byte (immediate, literal)

Load/Store Dual, Half, Signed Byte (register)

These instructions are under Extra load/store.

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11	10	9	8	7	6–5	4	3–0
!= 1111	0	0	0	P	U	0	W	o1	Rn	Rt	(0)	(0)	(0)	(0)	1	!= 00	1	Rm
cond																op2		

The following constraints also apply to this encoding: cond != 1111 && op2 != 00 && cond != 1111 && op2 != 00

Decode fields P	W	o1	op2	Instruction Details
0	0	0	01	STRH (register) — post-indexed
0	0	0	10	LDRD (register) — post-indexed
0	0	0	11	STRD (register) — post-indexed
0	0	1	01	LDRH (register) — post-indexed
0	0	1	10	LDRSB (register) — post-indexed
0	0	1	11	LDRSH (register) — post-indexed
0	1	0	01	STRHT
0	1	0	10	UNALLOCATED
0	1	0	11	UNALLOCATED
0	1	1	01	LDRHT
0	1	1	10	LDRSBT
0	1	1	11	LDRSHT
1		0	01	STRH (register) — pre-indexed
1		0	10	LDRD (register) — pre-indexed
1		0	11	STRD (register) — pre-indexed
1		1	01	LDRH (register) — pre-indexed
1		1	10	LDRSB (register) — pre-indexed
1		1	11	LDRSH (register) — pre-indexed

Load/Store Dual, Half, Signed Byte (immediate, literal)

These instructions are under Extra load/store.

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11–8	7	6–5	4	3–0
!= 1111	0	0	0	P	U	1	W	o1	Rn	Rt	imm4H	1	!= 00	1	imm4L
cond													op2		

The following constraints also apply to this encoding: cond != 1111 && op2 != 00 && cond != 1111 && op2 != 00

Decode fields P:W	o1	Rn	op2	Instruction Details
	0	1111	10	LDRD (literal)
!= 01	1	1111	01	LDRH (literal)
!= 01	1	1111	10	LDRSB (literal)
!= 01	1	1111	11	LDRSH (literal)
00	0	!= 1111	10	LDRD (immediate) — post-indexed
00	0		01	STRH (immediate) — post-indexed
00	0		11	STRD (immediate) — post-indexed
00	1	!= 1111	01	LDRH (immediate) — post-indexed
00	1	!= 1111	10	LDRSB (immediate) — post-indexed
00	1	!= 1111	11	LDRSH (immediate) — post-indexed
01	0	!= 1111	10	UNALLOCATED
01	0		01	STRHT
01	0		11	UNALLOCATED
01	1		01	LDRHT
01	1		10	LDRSBT
01	1		11	LDRSHT
10	0	!= 1111	10	LDRD (immediate) — offset
10	0		01	STRH (immediate) — offset

Decode fields P:W	o1	Rn	op2	Instruction Details
10	0		11	STRD (immediate) — offset
10	1	!= 1111	01	LDRH (immediate) — offset
10	1	!= 1111	10	LDRSB (immediate) — offset
10	1	!= 1111	11	LDRSH (immediate) — offset
11	0	!= 1111	10	LDRD (immediate) — pre-indexed
11	0		01	STRH (immediate) — pre-indexed
11	0		11	STRD (immediate) — pre-indexed
11	1	!= 1111	01	LDRH (immediate) — pre-indexed
11	1	!= 1111	10	LDRSB (immediate) — pre-indexed
11	1	!= 1111	11	LDRSH (immediate) — pre-indexed

Multiply and Accumulate

These instructions are under Data-processing and miscellaneous instructions.

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

The following constraints also apply to this encoding: cond != 1111 && cond != 1111

Decode fields opc	S	Instruction Details
000		MUL, MULS
001		MLA, MLAS
010	0	UMAAL
010	1	UNALLOCATED
011	0	MLS
011	1	UNALLOCATED
100		UMULL, UMULLS
101		UMLAL, UMLALS
110		SMULL, SMULLS
111		SMLAL, SMLALS

Synchronization primitives and Load-Acquire/Store-Release

These instructions are under Data-processing and miscellaneous instructions.

31 30 29 28	27 26 25 24	23	22 21 20 19 18 17 16 15 14 13 12	11 10	9 8	7 6 5 4	3 2 1 0
!= 1111	0001	op0		11		1001	

Decode fields op0	Instruction details
0	UNALLOCATED
1	Load/Store Exclusive and Load-Acquire/Store-Release

Load/Store Exclusive and Load-Acquire/Store-Release

These instructions are under Synchronization primitives and Load-Acquire/Store-Release.

31 30 29 28	27	26	25	24	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	size	L	Rn	xRd	(1)	(1)	ex	ord	1	0	0	1	xRt
cond																		

The following constraints also apply to this encoding: cond != 1111 && cond != 1111

Decode fields				Instruction Details
size	L	ex	ord	
00	0	0	0	STL
00	0	0	1	UNALLOCATED
00	0	1	0	STLEX
00	0	1	1	STREX
00	1	0	0	LDA
00	1	0	1	UNALLOCATED
00	1	1	0	LDAEX
00	1	1	1	LDREX
01	0	0		UNALLOCATED
01	0	1	0	STLEXD
01	0	1	1	STREXD
01	1	0		UNALLOCATED
01	1	1	0	LDAEXD
01	1	1	1	LDREXD
10	0	0	0	STLB
10	0	0	1	UNALLOCATED
10	0	1	0	STLEXB
10	0	1	1	STREXB
10	1	0	0	LDAB
10	1	0	1	UNALLOCATED
10	1	1	0	LDAEXB
10	1	1	1	LDREXB
11	0	0	0	STLH
11	0	0	1	UNALLOCATED
11	0	1	0	STLEXH
11	0	1	1	STREXH
11	1	0	0	LDAH
11	1	0	1	UNALLOCATED
11	1	1	0	LDAEXH
11	1	1	1	LDREXH

Miscellaneous

These instructions are under Data-processing and miscellaneous instructions.

31 30 29 28	27 26 25 24 23	22 21	20	19 18 17 16 15 14 13 12 11 10 9 8	7	6 5 4	3 2 1 0
!= 1111	00010	op0	0		0	op1	

Decode fields		Instruction details
op0	op1	
00	001	UNALLOCATED
00	010	UNALLOCATED
00	011	UNALLOCATED
00	110	UNALLOCATED

01	001	BX
01	010	BXJ
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.

31 30 29 28	27	26	25	24	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	opc	0	imm12	0	1	1	1	imm4
cond													

The following constraints also apply to this encoding: cond != 1111 && cond != 1111

Decode fields opc	Instruction Details
00	HLT
01	BKPT
10	HVC
11	SMC

Move special register (register)

These instructions are under Miscellaneous.

31 30 29 28	27	26	25	24	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	opc	0	mask	Rd	(0)	(0)	B	m	0	0	0	0	Rn
cond																		

The following constraints also apply to this encoding: cond != 1111 && cond != 1111

Decode fields opc	B	Instruction Details
x0	0	MRS
x0	1	MRS (Banked register)
x1	0	MSR (register)
x1	1	MSR (Banked register)

Cyclic Redundancy Check

These instructions are under Miscellaneous.

31 30 29 28	27	26	25	24	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)	C	(0)	0	1	0	0	Rm
cond																		

The following constraints also apply to this encoding: cond != 1111 && cond != 1111

Decode fields sz	C	Instruction Details	Feature
00	0	CRC32 — CRC32B	FEAT_CRC32
00	1	CRC32C — CRC32CB	FEAT_CRC32
01	0	CRC32 — CRC32H	FEAT_CRC32
01	1	CRC32C — CRC32CH	FEAT_CRC32
10	0	CRC32 — CRC32W	FEAT_CRC32
10	1	CRC32C — CRC32CW	FEAT_CRC32
11		CONSTRAINED UNPREDICTABLE	-

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.

31 30 29 28	27	26	25	24	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	opc	0	Rn	Rd	(0)	(0)	(0)	(0)	0	1	0	1	Rm
cond																		

The following constraints also apply to this encoding: cond != 1111 && cond != 1111

Decode fields opc	Instruction Details
00	QADD
01	QSUB
10	QDADD
11	QDSUB

Halfword Multiply and Accumulate

These instructions are under Data-processing and miscellaneous instructions.

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

The following constraints also apply to this encoding: cond != 1111 && cond != 1111

Decode fields opc	M	N	Instruction Details
00			SMLABB, SMLABT, SMLATB, SMLATT
01	0	0	SMLAWB, SMLAWT — SMLAWB
01	0	1	SMULWB, SMULWT — SMULWB

Decode fields			Instruction Details
opc	M	N	
01	1	0	SMLAWB, SMLAWT — SMLAWT
01	1	1	SMULWB, SMULWT — SMULWT
10			SMLALBB, SMLALBT, SMLALTB, SMLALTT
11			SMULBB, SMULBT, SMULTB, SMULTT

Data-processing register (immediate shift)

These instructions are under Data-processing and miscellaneous instructions.

31 30 29 28	27 26 25	24 23	22 21	20	19 18 17 16 15 14 13 12 11 10 9 8 7 6 5	4	3 2 1 0
!= 1111	000	op0		op1		0	

The following constraints also apply to this encoding: op0:op1 != 100

Decode fields		Instruction details
op0	op1	
0x		Integer Data Processing (three register, immediate shift)
10	1	Integer Test and Compare (two register, immediate shift)
11		Logical Arithmetic (three register, immediate shift)

Integer Data Processing (three register, immediate shift)

These instructions are under Data-processing register (immediate shift).

31 30 29 28	27	26	25	24	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	opc	S	Rn	Rd	imm5	stype	0	Rm
cond												

The following constraints also apply to this encoding: cond != 1111 && cond != 1111

Decode fields				Instruction Details
opc	S	Rn	imm5:stype	
000			!= 0000011	AND, ANDS (register) — shift or rotate by value
000			0000011	AND, ANDS (register) — rotate right with extend
001			!= 0000011	EOR, EORS (register) — shift or rotate by value
001			0000011	EOR, EORS (register) — rotate right with extend
010	0	!= 1101	!= 0000011	SUB, SUBS (register) — SUB, shift or rotate by value
010	0	!= 1101	0000011	SUB, SUBS (register) — SUB, rotate right with extend
010	0	1101	!= 0000011	SUB, SUBS (SP minus register) — SUB, shift or rotate by value
010	0	1101	0000011	SUB, SUBS (SP minus register) — SUB, rotate right with extend
010	1	!= 1101	!= 0000011	SUB, SUBS (register) — SUBS, shift or rotate by value
010	1	!= 1101	0000011	SUB, SUBS (register) — SUBS, rotate right with extend
010	1	1101	!= 0000011	SUB, SUBS (SP minus register) — SUBS, shift or rotate by value
010	1	1101	0000011	SUB, SUBS (SP minus register) — SUBS, rotate right with extend
011			!= 0000011	RSB, RSBS (register) — shift or rotate by value
011			0000011	RSB, RSBS (register) — rotate right with extend
100	0	!= 1101	!= 0000011	ADD, ADDS (register) — ADD, shift or rotate by value
100	0	!= 1101	0000011	ADD, ADDS (register) — ADD, rotate right with extend
100	0	1101	!= 0000011	ADD, ADDS (SP plus register) — ADD, shift or rotate by value
100	0	1101	0000011	ADD, ADDS (SP plus register) — ADD, rotate right with extend

Decode fields opc	S	Rn	imm5:stype	Instruction Details
100	1	!= 1101	!= 0000011	ADD, ADDS (register) — ADDS, shift or rotate by value
100	1	!= 1101	0000011	ADD, ADDS (register) — ADDS, rotate right with extend
100	1	1101	!= 0000011	ADD, ADDS (SP plus register) — ADDS, shift or rotate by value
100	1	1101	0000011	ADD, ADDS (SP plus register) — ADDS, rotate right with extend
101			!= 0000011	ADC, ADCS (register) — shift or rotate by value
101			0000011	ADC, ADCS (register) — rotate right with extend
110			!= 0000011	SBC, SBCS (register) — shift or rotate by value
110			0000011	SBC, SBCS (register) — rotate right with extend
111			!= 0000011	RSC, RSCS (register) — shift or rotate by value
111			0000011	RSC, RSCS (register) — rotate right with extend

Integer Test and Compare (two register, immediate shift)

These instructions are under Data-processing register (immediate shift).

31-28	27	26	25	24	23	22-21	20	19-16	15	14	13	12	11-7	6-5	4	3-0
!= 1111	0	0	0	1	0	opc	1	Rn	(0)	(0)	(0)	(0)	imm5	stype	0	Rm
cond																

The following constraints also apply to this encoding: cond != 1111 && cond != 1111

Decode fields opc	imm5:stype	Instruction Details
00	!= 0000011	TST (register) — shift or rotate by value
00	0000011	TST (register) — rotate right with extend
01	!= 0000011	TEQ (register) — shift or rotate by value
01	0000011	TEQ (register) — rotate right with extend
10	!= 0000011	CMP (register) — shift or rotate by value
10	0000011	CMP (register) — rotate right with extend
11	!= 0000011	CMN (register) — shift or rotate by value
11	0000011	CMN (register) — rotate right with extend

Logical Arithmetic (three register, immediate shift)

These instructions are under Data-processing register (immediate shift).

31-28	27	26	25	24	23	22-21	20	19-16	15-12	11-7	6-5	4	3-0
!= 1111	0	0	0	1	1	opc	S	Rn	Rd	imm5	stype	0	Rm
cond													

The following constraints also apply to this encoding: cond != 1111 && cond != 1111

Decode fields opc	imm5:stype	Instruction Details
00	!= 0000011	ORR, ORRS (register) — shift or rotate by value
00	0000011	ORR, ORRS (register) — rotate right with extend
01	!= 0000011	MOV, MOVS (register) — shift or rotate by value
01	0000011	MOV, MOVS (register) — rotate right with extend
10	!= 0000011	BIC, BICS (register) — shift or rotate by value
10	0000011	BIC, BICS (register) — rotate right with extend
11	!= 0000011	MVN, MVNS (register) — shift or rotate by value

Decode fields opc	imm5:stype	Instruction Details
11	0000011	MVN, MVNS (register) — rotate right with extend

Data-processing register (register shift)

These instructions are under Data-processing and miscellaneous instructions.

31 30 29 28	27 26 25	24 23	22 21	20	19 18 17 16 15 14 13 12 11 10 9 8	7	6 5	4	3 2 1 0
!= 1111	000	op0		op1		0		1	

The following constraints also apply to this encoding: op0:op1 != 100

Decode fields op0	op1	Instruction details
0x		Integer Data Processing (three register, register shift)
10	1	Integer Test and Compare (two register, register shift)
11		Logical Arithmetic (three register, register shift)

Integer Data Processing (three register, register shift)

These instructions are under Data-processing register (register shift).

31 30 29 28	27	26	25	24	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	opc	S	Rn	Rd	Rs	0	stype	1	Rm
cond													

The following constraints also apply to this encoding: cond != 1111 && cond != 1111

Decode fields opc	Instruction Details
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)

Integer Test and Compare (two register, register shift)

These instructions are under Data-processing register (register shift).

31 30 29 28	27	26	25	24	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	opc	1	Rn	(0)	(0)	(0)	(0)	Rs	0	stype	1	Rm
cond																	

The following constraints also apply to this encoding: cond != 1111 && cond != 1111

Decode fields opc	Instruction Details
00	TST (register-shifted register)
01	TEQ (register-shifted register)

Decode fields opc	Instruction Details
10	CMP (register-shifted register)
11	CMN (register-shifted register)

Logical Arithmetic (three register, register shift)

These instructions are under Data-processing register (register shift).

31–28	27	26	25	24	23	22–21	20	19–16	15–12	11–8	7	6–5	4	3–0
!= 1111	0	0	0	1	1	opc	S	Rn	Rd	Rs	0	stype	1	Rm
cond														

The following constraints also apply to this encoding: cond != 1111 && cond != 1111

Decode fields opc	Instruction Details
00	ORR, ORRS (register-shifted register)
01	MOV, MOVS (register-shifted register)
10	BIC, BICS (register-shifted register)
11	MVN, MVNS (register-shifted register)

Data-processing immediate

These instructions are under Data-processing and miscellaneous instructions.

31–28	27–25	24–23	22	21–20	19–0
!= 1111	001	op0		op1	

Decode fields op0	op1	Instruction details
0x		Integer Data Processing (two register and immediate)
10	00	Move Halfword (immediate)
10	10	Move Special Register and Hints (immediate)
10	x1	Integer Test and Compare (one register and immediate)
11		Logical Arithmetic (two register and immediate)

Integer Data Processing (two register and immediate)

These instructions are under Data-processing immediate.

31–28	27	26	25	24	23–21	20	19–16	15–12	11–0
!= 1111	0	0	1	0	opc	S	Rn	Rd	imm12
cond									

The following constraints also apply to this encoding: cond != 1111 && cond != 1111

Decode fields opc	S	Rn	Instruction Details
000			AND, ANDS (immediate)
001			EOR, EORS (immediate)
010	0	!= 11x1	SUB, SUBS (immediate) — SUB
010	0	1101	SUB, SUBS (SP minus immediate) — SUB
010	0	1111	ADR — A2

Decode fields opc	S	Rn	Instruction Details
010	1	!= 1101	SUB, SUBS (immediate) — SUBS
010	1	1101	SUB, SUBS (SP minus immediate) — SUBS
011			RSB, RSBS (immediate)
100	0	!= 11x1	ADD, ADDS (immediate) — ADD
100	0	1101	ADD, ADDS (SP plus immediate) — ADD
100	0	1111	ADR — A1
100	1	!= 1101	ADD, ADDS (immediate) — ADDS
100	1	1101	ADD, ADDS (SP plus immediate) — ADDS
101			ADC, ADCS (immediate)
110			SBC, SBCS (immediate)
111			RSC, RSCS (immediate)

Move Halfword (immediate)

These instructions are under Data-processing immediate.

31 30 29 28	27	26	25	24	23	22	21	20	19 18 17 16	15 14 13 12	11 10 9 8 7 6 5 4 3 2 1 0
!= 1111	0	0	1	1	0	H	0	0	imm4	Rd	imm12
cond											

The following constraints also apply to this encoding: cond != 1111 && cond != 1111

Decode fields H	Instruction Details
0	MOV, MOVS (immediate)
1	MOVT

Move Special Register and Hints (immediate)

These instructions are under Data-processing immediate.

31 30 29 28	27	26	25	24	23	22	21	20	19 18 17 16	15	14	13	12	11 10 9 8 7 6 5 4 3 2 1 0
!= 1111	0	0	1	1	0	R	1	0	imm4	(1)	(1)	(1)	(1)	imm12
cond														

The following constraints also apply to this encoding: cond != 1111 && cond != 1111

Decode fields R:imm4	imm12	Instruction Details	Feature
!= 00000		MSR (immediate)	-
00000	xxxx00000000	NOP	-
00000	xxxx00000001	YIELD	-
00000	xxxx00000010	WFE	-
00000	xxxx00000011	WFI	-
00000	xxxx00000100	SEV	-
00000	xxxx00000101	SEVL	-
00000	xxxx0000011x	Reserved hint, behaves as NOP	-
00000	xxxx00001xxx	Reserved hint, behaves as NOP	-
00000	xxxx00010000	ESB	FEAT_RAS
00000	xxxx00010001	Reserved hint, behaves as NOP	-
00000	xxxx00010010	TSB CSYNC	FEAT_TRF

Decode fields R:imm4	imm12	Instruction Details	Feature
00000	xxxx00010011	Reserved hint, behaves as NOP	-
00000	xxxx00010100	CSDB	-
00000	xxxx00010101	Reserved hint, behaves as NOP	-
00000	xxxx00010110	CLRBHB	FEAT_CLRBHB
00000	xxxx00010111	Reserved hint, behaves as NOP	-
00000	xxxx00011xxx	Reserved hint, behaves as NOP	-
00000	xxxx001xxxxx	Reserved hint, behaves as NOP	-
00000	xxxx01xxxxxx	Reserved hint, behaves as NOP	-
00000	xxxx10xxxxxx	Reserved hint, behaves as NOP	-
00000	xxxx110xxxxx	Reserved hint, behaves as NOP	-
00000	xxxx1110xxxx	Reserved hint, behaves as NOP	-
00000	xxxx1111xxxx	DBG	-

Integer Test and Compare (one register and immediate)

These instructions are under Data-processing immediate.

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

The following constraints also apply to this encoding: cond != 1111 && cond != 1111

Decode fields opc	Instruction Details
00	TST (immediate)
01	TEQ (immediate)
10	CMP (immediate)
11	CMN (immediate)

Logical Arithmetic (two register and immediate)

These instructions are under Data-processing immediate.

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

The following constraints also apply to this encoding: cond != 1111 && cond != 1111

Decode fields opc	Instruction Details
00	ORR, ORRS (immediate)
01	MOV, MOVS (immediate)
10	BIC, BICS (immediate)
11	MVN, MVNS (immediate)

Load/Store Word, Unsigned Byte (immediate, literal)

31 30 29 28	27	26	25	24	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	o2	W	o1	Rn	Rt	imm12
cond											

The following constraints also apply to this encoding: cond != 1111 && cond != 1111

Decode fields				Instruction Details
P:W	o2	o1	Rn	
!= 01	0	1	1111	LDR (literal)
!= 01	1	1	1111	LDRB (literal)
00	0	0		STR (immediate) — post-indexed
00	0	1	!= 1111	LDR (immediate) — post-indexed
00	1	0		STRB (immediate) — post-indexed
00	1	1	!= 1111	LDRB (immediate) — post-indexed
01	0	0		STRT
01	0	1		LDRT
01	1	0		STRBT
01	1	1		LDRBT
10	0	0		STR (immediate) — offset
10	0	1	!= 1111	LDR (immediate) — offset
10	1	0		STRB (immediate) — offset
10	1	1	!= 1111	LDRB (immediate) — offset
11	0	0		STR (immediate) — pre-indexed
11	0	1	!= 1111	LDR (immediate) — pre-indexed
11	1	0		STRB (immediate) — pre-indexed
11	1	1	!= 1111	LDRB (immediate) — pre-indexed

Load/Store Word, Unsigned Byte (register)

31 30 29 28	27 26 25	24	23	22	21	20	19 18 17 16	15 14 13 12	11 10 9 8 7	6 5	4	3 2 1 0
!= 1111	0 1 1	P	U	o2	W	o1	Rn	Rt	imm5	stype	0	Rm
cond												

The following constraints also apply to this encoding: cond != 1111 && cond != 1111

Decode fields				Instruction Details
P	o2	W	o1	
0	0	0	0	STR (register) — post-indexed
0	0	0	1	LDR (register) — post-indexed
0	0	1	0	STRT
0	0	1	1	LDRT
0	1	0	0	STRB (register) — post-indexed
0	1	0	1	LDRB (register) — post-indexed
0	1	1	0	STRBT
0	1	1	1	LDRBT
1	0		0	STR (register) — pre-indexed
1	0		1	LDR (register) — pre-indexed
1	1		0	STRB (register) — pre-indexed
1	1		1	LDRB (register) — pre-indexed

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
!= 1111	011	op0		op1	1	

Decode fields op0	op1	Instruction details
00xxx		Parallel Arithmetic
01000	101	SEL
01000	001	UNALLOCATED
01000	xx0	PKHBT, PKHTB
01001	x01	UNALLOCATED
01001	xx0	UNALLOCATED
0110x	x01	UNALLOCATED
0110x	xx0	UNALLOCATED
01x10	001	Saturate 16-bit
01x10	101	UNALLOCATED
01x11	x01	Reverse Bit/Byte
01x1x	xx0	Saturate 32-bit
01xxx	111	UNALLOCATED
01xxx	011	Extend and Add
10xxx		Signed multiply, Divide
11000	000	Unsigned Sum of Absolute Differences
11000	100	UNALLOCATED
11001	x00	UNALLOCATED
1101x	x00	UNALLOCATED
110xx	111	UNALLOCATED
1110x	111	UNALLOCATED
1110x	x00	Bitfield Insert
11110	111	UNALLOCATED
11111	111	Permanently UNDEFINED
1111x	x00	UNALLOCATED
11x0x	x10	UNALLOCATED
11x1x	x10	Bitfield Extract
11xxx	011	UNALLOCATED
11xxx	x01	UNALLOCATED

Parallel Arithmetic

These instructions are under Media instructions.

31 30 29 28	27	26	25	24	23	22 21 20	19 18 17 16	15 14 13 12	11	10	9	8	7	6 5	4	3 2 1 0
!= 1111	0	1	1	0	0	op1	Rn	Rd	(1)	(1)	(1)	(1)	B	op2	1	Rm
cond																

The following constraints also apply to this encoding: cond != 1111 && cond != 1111

Decode fields op1	B	op2	Instruction Details
000			UNALLOCATED
001	0	00	SADD16
001	0	01	SASX
001	0	10	SSAX
001	0	11	SSUB16
001	1	00	SADD8
001	1	01	UNALLOCATED

Decode fields			Instruction Details
op1	B	op2	
001	1	10	UNALLOCATED
001	1	11	SSUB8
010	0	00	QADD16
010	0	01	QASX
010	0	10	QSAX
010	0	11	QSUB16
010	1	00	QADD8
010	1	01	UNALLOCATED
010	1	10	UNALLOCATED
010	1	11	QSUB8
011	0	00	SHADD16
011	0	01	SHASX
011	0	10	SHSAX
011	0	11	SHSUB16
011	1	00	SHADD8
011	1	01	UNALLOCATED
011	1	10	UNALLOCATED
011	1	11	SHSUB8
100			UNALLOCATED
101	0	00	UADD16
101	0	01	UASX
101	0	10	USAX
101	0	11	USUB16
101	1	00	UADD8
101	1	01	UNALLOCATED
101	1	10	UNALLOCATED
101	1	11	USUB8
110	0	00	UQADD16
110	0	01	UQASX
110	0	10	UQSAX
110	0	11	UQSUB16
110	1	00	UQADD8
110	1	01	UNALLOCATED
110	1	10	UNALLOCATED
110	1	11	UQSUB8
111	0	00	UHADD16
111	0	01	UHASX
111	0	10	UHSAX
111	0	11	UHSUB16
111	1	00	UHADD8
111	1	01	UNALLOCATED
111	1	10	UNALLOCATED
111	1	11	UHSUB8

Saturate 16-bit

These instructions are under Media instructions.

31-28	27	26	25	24	23	22	21	20	19-16	15-12	11	10	9	8	7	6	5	4	3-0
!= 1111	0	1	1	0	1	U	1	0	sat_imm	Rd	(1)	(1)	(1)	(1)	0	0	1	1	Rn
cond																			

The following constraints also apply to this encoding: cond != 1111 && cond != 1111

Decode fields U	Instruction Details
0	SSAT16
1	USAT16

Reverse Bit/Byte

These instructions are under Media instructions.

31-28	27	26	25	24	23	22	21	20	19	18	17	16	15-12	11	10	9	8	7	6	5	4	3-0
!= 1111	0	1	1	0	1	o1	1	1	(1)	(1)	(1)	(1)	Rd	(1)	(1)	(1)	(1)	o2	0	1	1	Rm
cond																						

The following constraints also apply to this encoding: cond != 1111 && cond != 1111

Decode fields o1	 o2	Instruction Details
0	0	REV
0	1	REV16
1	0	RBIT
1	1	REVSH

Saturate 32-bit

These instructions are under Media instructions.

31-28	27	26	25	24	23	22	21	20-16	15-12	11-7	6	5	4	3-0
!= 1111	0	1	1	0	1	U	1	sat_imm	Rd	imm5	sh	0	1	Rn
cond														

The following constraints also apply to this encoding: cond != 1111 && cond != 1111

Decode fields U	Instruction Details
0	SSAT
1	USAT

Extend and Add

These instructions are under Media instructions.

31-28	27	26	25	24	23	22	21-20	19-16	15-12	11-10	9	8	7	6	5	4	3-0
!= 1111	0	1	1	0	1	U	op	Rn	Rd	rotate	(0)	(0)	0	1	1	1	Rm
cond																	

The following constraints also apply to this encoding: cond != 1111 && cond != 1111

Decode fields U	 op	 Rn	Instruction Details
0	00	!= 1111	SXTAB16

Decode fields			Instruction Details
U	op	Rn	
0	00	1111	SXTB16
0	10	!= 1111	SXTAB
0	10	1111	SXTB
0	11	!= 1111	SXTAH
0	11	1111	SXTH
1	00	!= 1111	UXTAB16
1	00	1111	UXTB16
1	10	!= 1111	UXTAB
1	10	1111	UXTB
1	11	!= 1111	UXTAH
1	11	1111	UXTH

Signed multiply, Divide

These instructions are under Media instructions.

31 30 29 28	27	26	25	24	23	22 21 20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5	4	3 2 1 0
!= 1111	0	1	1	1	0	op1	Rd	Ra	Rm	op2	1	Rn
cond												

The following constraints also apply to this encoding: cond != 1111 && cond != 1111

Decode fields			Instruction Details
op1	Ra	op2	
000	!= 1111	000	SMLAD, SMLADX — SMLAD
000	!= 1111	001	SMLAD, SMLADX — SMLADX
000	!= 1111	010	SMLSD, SMLSDX — SMLSD
000	!= 1111	011	SMLSD, SMLSDX — SMLSDX
000		1xx	UNALLOCATED
000	1111	000	SMUAD, SMUADX — SMUAD
000	1111	001	SMUAD, SMUADX — SMUADX
000	1111	010	SMUSD, SMUSDX — SMUSD
000	1111	011	SMUSD, SMUSDX — SMUSDX
001		000	SDIV
001		!= 000	UNALLOCATED
010			UNALLOCATED
011		000	UDIV
011		!= 000	UNALLOCATED
100		000	SMLALD, SMLALDX — SMLALD
100		001	SMLALD, SMLALDX — SMLALDX
100		010	SMLSLD, SMLSLDX — SMLSLD
100		011	SMLSLD, SMLSLDX — SMLSLDX
100		1xx	UNALLOCATED
101	!= 1111	000	SMMLA, SMMLAR — SMMLA
101	!= 1111	001	SMMLA, SMMLAR — SMMLAR
101		01x	UNALLOCATED
101		10x	UNALLOCATED
101		110	SMMLS, SMMLSR — SMMLS
101		111	SMMLS, SMMLSR — SMMLSR

Decode fields op1	Ra	op2	Instruction Details
101	1111	000	SMMUL, SMMULR — SMMUL
101	1111	001	SMMUL, SMMULR — SMMULR
11x			UNALLOCATED

Unsigned Sum of Absolute Differences

These instructions are under Media instructions.

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

The following constraints also apply to this encoding: cond != 1111 && cond != 1111

Decode fields Ra	Instruction Details
!= 1111	USADA8
1111	USAD8

Bitfield Insert

These instructions are under Media instructions.

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

The following constraints also apply to this encoding: cond != 1111 && cond != 1111

Decode fields Rn	Instruction Details
!= 1111	BFI
1111	BFC

Permanently UNDEFINED

These instructions are under Media instructions.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	1	1	1	1	1	1	1	imm12												1	1	1	1	imm4			
cond																															

The following constraints also apply to this encoding: cond != 1111 && cond != 1111

Decode fields cond	Instruction Details
0xxx	UNALLOCATED
10xx	UNALLOCATED
110x	UNALLOCATED
1110	UDF

Bitfield Extract

These instructions are under Media instructions.

31–28	27	26	25	24	23	22	21	20–16	15–12	11–7	6	5	4	3–0
!= 1111	0	1	1	1	1	U	1	widthm1	Rd	lsb	1	0	1	Rn
cond														

The following constraints also apply to this encoding: cond != 1111 && cond != 1111

Decode fields U	Instruction Details
0	SBFX
1	UBFX

Branch, branch with link, and block data transfer

These instructions are under the top-level.

31–28	27–26	25	24–0
cond	10	op0	

Decode fields cond	op0	Instruction details
1111	0	Exception Save/Restore
!= 1111	0	Load/Store Multiple
	1	Branch (immediate)

Exception Save/Restore

These instructions are under Branch, branch with link, and block data transfer.

31	30	29	28	27	26	25	24	23	22	21	20	19–16	15–5	4–0
1	1	1	1	1	0	0	P	U	S	W	L	Rn	op	mode

Decode fields P	U	S	L	Instruction Details
		0	0	UNALLOCATED
0	0	0	1	RFE, RFEDA, RFEDB, RFEIA, RFEIB — Decrement After
0	0	1	0	SRS, SRSDA, SRSDB, SRSIA, SRSIB — Decrement After
0	1	0	1	RFE, RFEDA, RFEDB, RFEIA, RFEIB — Increment After
0	1	1	0	SRS, SRSDA, SRSDB, SRSIA, SRSIB — Increment After
1	0	0	1	RFE, RFEDA, RFEDB, RFEIA, RFEIB — Decrement Before
1	0	1	0	SRS, SRSDA, SRSDB, SRSIA, SRSIB — Decrement Before
		1	1	UNALLOCATED
1	1	0	1	RFE, RFEDA, RFEDB, RFEIA, RFEIB — Increment Before
1	1	1	0	SRS, SRSDA, SRSDB, SRSIA, SRSIB — Increment Before

Load/Store Multiple

These instructions are under Branch, branch with link, and block data transfer.

31–28	27	26	25	24	23	22	21	20	19–16	15–0
!= 1111	1	0	0	P	U	op	W	L	Rn	register_list
cond										

The following constraints also apply to this encoding: cond != 1111 && cond != 1111

Decode fields P	U	op	L	register_list	Instruction Details
0	0	0	0		STMDA, STMED
0	0	0	1		LDMDA, LDMFA
0	1	0	0		STM, STMIA, STMEA
0	1	0	1		LDM, LDMIA, LDMFD
		1	0		STM (User registers)
1	0	0	0		STMDB, STMFD
1	0	0	1		LDMDB, LDMEA
		1	1	0xxxxxxxxxxxxxxx	LDM (User registers)
1	1	0	0		STMIB, STMFA
1	1	0	1		LDMIB, LDMED
		1	1	1xxxxxxxxxxxxxxx	LDM (exception return)

Branch (immediate)

These instructions are under Branch, branch with link, and block data transfer.

31 30 29 28	27 26 25	24	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	1 0 1	H	imm24

Decode fields cond	H	Instruction Details
!= 1111	0	B
!= 1111	1	BL, BLX (immediate) — A1
1111		BL, BLX (immediate) — A2

System register access, Advanced SIMD, floating-point, and Supervisor call

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
cond	11	op0		op1		op2	

Decode fields cond	op0	op1	op2	Instruction details
	0x	0x		UNALLOCATED
	10	0x		UNALLOCATED
	11			Supervisor call
1111	!= 11	1x		Unconditional Advanced SIMD and floating-point instructions
!= 1111	0x	1x		Advanced SIMD and System register load/store and 64-bit move
!= 1111	10	1x	1	Advanced SIMD and System register 32-bit move
!= 1111	10	10	0	Floating-point data-processing
!= 1111	10	11	0	UNALLOCATED

Supervisor call

These instructions are under System register access, Advanced SIMD, floating-point, and Supervisor call.

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

Decode fields **Instruction details**

cond	
1111	UNALLOCATED
!= 1111	SVC

Unconditional Advanced SIMD and floating-point instructions

These instructions are under System register access, Advanced SIMD, floating-point, and Supervisor call.

31–26	25–23	22	21–16	15–12	11	10	9–8	7	6	5	4	3–0
111111	op0		op1		1	op2	op3		op4		op5	

The following constraints also apply to this encoding: op0<2:1> != 11

Decode fields						Instruction details
op0	op1	op2	op3	op4	op5	
0xx			0x			Advanced SIMD three registers of the same length extension
100		0	!= 00	0	0	Floating-point conditional select
101	00xxxx	0	!= 00		0	Floating-point minNum/maxNum
101	110000	0	!= 00	1	0	Floating-point extraction and insertion
101	111xxx	0	!= 00	1	0	Floating-point directed convert to integer
10x		0	00			Advanced SIMD and floating-point multiply with accumulate
10x		1	0x			Advanced SIMD and floating-point dot product

Advanced SIMD three registers of the same length extension

These instructions are under Unconditional Advanced SIMD and floating-point instructions.

31	30	29	28	27	26	25	24–23	22	21–20	19–16	15–12	11	10	9	8	7	6	5	4	3–0
1	1	1	1	1	1	0	op1	D	op2	Vn	Vd	1	op3	0	op4	N	Q	M	U	Vm

Decode fields						Instruction Details	Feature
op1	op2	op3	op4	Q	U		
x1	0x	0	0	0	0	VCADD — 64-bit SIMD vector	FEAT_FCMA
x1	0x	0	0	0	1	UNALLOCATED	-
x1	0x	0	0	1	0	VCADD — 128-bit SIMD vector	FEAT_FCMA
x1	0x	0	0	1	1	UNALLOCATED	-
00	0x	0	0			UNALLOCATED	-
00	0x	0	1			UNALLOCATED	-
00	00	1	0	0	0	UNALLOCATED	-
00	00	1	0	0	1	UNALLOCATED	-
00	00	1	0	1	0	VMMLA	FEAT_AA32BF16
00	00	1	0	1	1	UNALLOCATED	-
00	00	1	1	0	0	VDOT (vector) — 64-bit SIMD vector	FEAT_AA32BF16
00	00	1	1	0	1	UNALLOCATED	-
00	00	1	1	1	0	VDOT (vector) — 128-bit SIMD vector	FEAT_AA32BF16
00	00	1	1	1	1	UNALLOCATED	-
00	01	1	0			UNALLOCATED	-

Decode fields op1	op2	op3	op4	Q	U	Instruction Details	Feature
00	01	1	1			UNALLOCATED	-
00	10	0	0		1	VFMAL (vector)	FEAT_FHM
00	10	0	1			UNALLOCATED	-
00	10	1	0	0		UNALLOCATED	-
00	10	1	0	1	0	VSMMLA	FEAT_AA32I8MM
00	10	1	0	1	1	VUMMLA	FEAT_AA32I8MM
00	10	1	1	0	0	VSDOT (vector) — 64-bit SIMD vector	FEAT_DotProd
00	10	1	1	0	1	VUDOT (vector) — 64-bit SIMD vector	FEAT_DotProd
00	10	1	1	1	0	VSDOT (vector) — 128-bit SIMD vector	FEAT_DotProd
00	10	1	1	1	1	VUDOT (vector) — 128-bit SIMD vector	FEAT_DotProd
00	11	0	0		1	VFMAB, VFMAT (BFloat16, vector)	FEAT_AA32BF16
00	11	0	1			UNALLOCATED	-
00	11	1	0			UNALLOCATED	-
00	11	1	1			UNALLOCATED	-
01	10	0	0		1	VFMSL (vector)	FEAT_FHM
01	10	0	1			UNALLOCATED	-
01	10	1	0	0		UNALLOCATED	-
01	10	1	0	1	0	VUSMMLA	FEAT_AA32I8MM
01	10	1	0	1	1	UNALLOCATED	-
01	10	1	1	0	0	VUSDOT (vector) — 64-bit SIMD vector	FEAT_AA32I8MM
01	10	1	1		1	UNALLOCATED	-
01	10	1	1	1	0	VUSDOT (vector) — 128-bit SIMD vector	FEAT_AA32I8MM
01	11	0	1			UNALLOCATED	-
01	11	1	0			UNALLOCATED	-
01	11	1	1			UNALLOCATED	-
	1x	0	0		0	VCMLA	FEAT_FCMA
10	11	0	1			UNALLOCATED	-
10	11	1	0			UNALLOCATED	-
10	11	1	1			UNALLOCATED	-
11	11	0	1			UNALLOCATED	-
11	11	1	0			UNALLOCATED	-
11	11	1	1			UNALLOCATED	-

Floating-point conditional select

These instructions are under Unconditional Advanced SIMD and floating-point instructions.

31	30	29	28	27	26	25	24	23	22	21 20	19 18 17 16	15 14 13 12	11	10	9 8	7	6	5	4	3 2 1 0
1	1	1	1	1	1	1	0	0	D	cc	Vn	Vd	1	0	!= 00	N	0	M	0	Vm
															size					

The following constraints also apply to this encoding: size != 00 && size != 00

Decode fields size	Instruction Details	Feature
01	VSELEQ, VSELGE, VSELGT, VSELVS — half-precision scalar	FEAT_FP16
10	VSELEQ, VSELGE, VSELGT, VSELVS — single-precision scalar	-
11	VSELEQ, VSELGE, VSELGT, VSELVS — double-precision scalar	-

Floating-point minNum/maxNum

These instructions are under Unconditional Advanced SIMD and floating-point instructions.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	0	1	D	0	0	Vn				Vd				1	0	!= 00		N	op	M	0	Vm			
																						size									

The following constraints also apply to this encoding: size != 00 && size != 00

Decode fields size	op	Instruction Details	Feature
01	0	VMAXNM — half-precision scalar	FEAT_FP16
01	1	VMINNM — half-precision scalar	FEAT_FP16
10	0	VMAXNM — single-precision scalar	-
10	1	VMINNM — single-precision scalar	-
11	0	VMAXNM — double-precision scalar	-
11	1	VMINNM — double-precision scalar	-

Floating-point extraction and insertion

These instructions are under Unconditional Advanced SIMD and floating-point instructions.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	0	1	D	1	1	0	0	0	0	Vd				1	0	!= 00		op	1	M	0	Vm			
																						size									

The following constraints also apply to this encoding: size != 00 && size != 00

Decode fields size	op	Instruction Details	Feature
01		UNALLOCATED	-
10	0	VMOVX	FEAT_FP16
10	1	VINS	FEAT_FP16
11		UNALLOCATED	-

Floating-point directed convert to integer

These instructions are under Unconditional Advanced SIMD and floating-point instructions.

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

Decode fields o1	RM	size	op	Instruction Details	Feature
0		!= 00	1	UNALLOCATED	-
0	00	01	0	VRINTA (floating-point) — half-precision scalar	FEAT_FP16
0	00	10	0	VRINTA (floating-point) — single-precision scalar	-
0	00	11	0	VRINTA (floating-point) — double-precision scalar	-
0	01	01	0	VRINTN (floating-point) — half-precision scalar	FEAT_FP16
0	01	10	0	VRINTN (floating-point) — single-precision scalar	-
0	01	11	0	VRINTN (floating-point) — double-precision scalar	-

Decode fields				Instruction Details	Feature
o1	RM	size	op		
0	10	01	0	VRINTP (floating-point) — half-precision scalar	FEAT_FP16
0	10	10	0	VRINTP (floating-point) — single-precision scalar	-
0	10	11	0	VRINTP (floating-point) — double-precision scalar	-
0	11	01	0	VRINTM (floating-point) — half-precision scalar	FEAT_FP16
0	11	10	0	VRINTM (floating-point) — single-precision scalar	-
0	11	11	0	VRINTM (floating-point) — double-precision scalar	-
1	00	01		VCVTA (floating-point) — half-precision scalar	FEAT_FP16
1	00	10		VCVTA (floating-point) — single-precision scalar	-
1	00	11		VCVTA (floating-point) — double-precision scalar	-
1	01	01		VCVTN (floating-point) — half-precision scalar	FEAT_FP16
1	01	10		VCVTN (floating-point) — single-precision scalar	-
1	01	11		VCVTN (floating-point) — double-precision scalar	-
1	10	01		VCVTP (floating-point) — half-precision scalar	FEAT_FP16
1	10	10		VCVTP (floating-point) — single-precision scalar	-
1	10	11		VCVTP (floating-point) — double-precision scalar	-
1	11	01		VCVTM (floating-point) — half-precision scalar	FEAT_FP16
1	11	10		VCVTM (floating-point) — single-precision scalar	-
1	11	11		VCVTM (floating-point) — double-precision scalar	-

Advanced SIMD and floating-point multiply with accumulate

These instructions are under Unconditional Advanced SIMD and floating-point instructions.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	0	op1	D	op2		Vn				Vd				1	0	0	0	N	Q	M	U	Vm			

Decode fields				Instruction Details	Feature
op1	op2	Q	U		
0			0	VCMLA (by element) — 128-bit SIMD vector of half-precision floating-point	FEAT_FCMA
0	00		1	VFMAL (by scalar)	FEAT_FHM
0	01		1	VFMSL (by scalar)	FEAT_FHM
0	10		1	UNALLOCATED	-
0	11		1	VFMAB, VFMAT (BFloat16, by scalar)	FEAT_AA32BF16
1		0	0	VCMLA (by element) — 64-bit SIMD vector of single-precision floating-point	FEAT_FCMA
1			1	UNALLOCATED	-
1		1	0	VCMLA (by element) — 128-bit SIMD vector of single-precision floating-point	FEAT_FCMA

Advanced SIMD and floating-point dot product

These instructions are under Unconditional Advanced SIMD and floating-point instructions.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	0	op1	D	op2		Vn				Vd				1	1	0	op4	N	Q	M	U	Vm			

Decode fields					Instruction Details	Feature
op1	op2	op4	Q	U		
0	00	0			UNALLOCATED	-
0	00	1	0	0	VDOT (by element) — 64-bit SIMD vector	FEAT_AA32BF16

Decode fields					Instruction Details	Feature
op1	op2	op4	Q	U		
0	00	1		1	UNALLOCATED	-
0	00	1	1	0	VDOT (by element) — 128-bit SIMD vector	FEAT_AA32BF16
0	01	0			UNALLOCATED	-
0	10	0			UNALLOCATED	-
0	10	1	0	0	VSDOT (by element) — 64-bit SIMD vector	FEAT_DotProd
0	10	1	0	1	VUDOT (by element) — 64-bit SIMD vector	FEAT_DotProd
0	10	1	1	0	VSDOT (by element) — 128-bit SIMD vector	FEAT_DotProd
0	10	1	1	1	VUDOT (by element) — 128-bit SIMD vector	FEAT_DotProd
0	11				UNALLOCATED	-
1		0			UNALLOCATED	-
1	00	1	0	0	VUSDOT (by element) — 64-bit SIMD vector	FEAT_AA32I8MM
1	00	1	0	1	VSUDOT (by element) — 64-bit SIMD vector	FEAT_AA32I8MM
1	00	1	1	0	VUSDOT (by element) — 128-bit SIMD vector	FEAT_AA32I8MM
1	00	1	1	1	VSUDOT (by element) — 128-bit SIMD vector	FEAT_AA32I8MM
1	01	1			UNALLOCATED	-
1	1x	1			UNALLOCATED	-

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.

31 30 29 28	27 26 25	24 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	110	op0		1	op1	

Decode fields		Instruction details
op0	op1	
00x0	0x	Advanced SIMD and floating-point 64-bit move
00x0	11	System register 64-bit move
!= 00x0	0x	Advanced SIMD and floating-point load/store
!= 00x0	11	System register load/store
	10	UNALLOCATED

Advanced SIMD and floating-point 64-bit move

These instructions are under Advanced SIMD and System register load/store and 64-bit move.

31 30 29 28	27 26 25 24 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	D	0	op	Rt2	Rt	1 0	size	opc2	M	o3	Vm
cond												

The following constraints also apply to this encoding: cond != 1111 && cond != 1111

Decode fields					Instruction Details
D	op	size	opc2	o3	
0					UNALLOCATED
1				0	UNALLOCATED
1		0x	00	1	UNALLOCATED
1			01		UNALLOCATED
1	0	10	00	1	VMOV (between two general-purpose registers and two single-precision registers) — from general-purpose registers

Decode fields D	op	size	opc2	o3	Instruction Details
1	0	11	00	1	VMOV (between two general-purpose registers and a doubleword floating-point register) — from general-purpose registers
1			1x		UNALLOCATED
1	1	10	00	1	VMOV (between two general-purpose registers and two single-precision registers) — to general-purpose registers
1	1	11	00	1	VMOV (between two general-purpose registers and a doubleword floating-point register) — to general-purpose registers

System register 64-bit move

These instructions are under Advanced SIMD and System register load/store and 64-bit move.

31 30 29 28	27 26 25 24 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	D	0	L	Rt2	Rt	1 1 1	cp15	opc1	CRm
cond										

The following constraints also apply to this encoding: cond != 1111 && cond != 1111

Decode fields D	L	Instruction Details
0		UNALLOCATED
1	0	MCRR
1	1	MRRC

Advanced SIMD and floating-point load/store

These instructions are under Advanced SIMD and System register load/store and 64-bit move.

31 30 29 28	27 26 25	24	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	L	Rn	Vd	1 0	size	imm8
cond											

The following constraints also apply to this encoding: cond != 1111 && P:U:D:W != 00x0 && cond != 1111

Decode fields P	U	W	L	Rn	size	imm8	Instruction Details	Feature
0	0	1					UNALLOCATED	-
0	1				0x		UNALLOCATED	-
0	1		0		10		VSTM, VSTMDB, VSTMIA — single-precision scalar	-
0	1		0		11	xxxxxxx0	VSTM, VSTMDB, VSTMIA — double-precision scalar	-
0	1		0		11	xxxxxxx1	FSTMDBX, FSTMIAX — Increment After	-
0	1		1		10		VLDM, VLDMDB, VLDMIA — single-precision scalar	-
0	1		1		11	xxxxxxx0	VLDM, VLDMDB, VLDMIA — double-precision scalar	-
0	1		1		11	xxxxxxx1	FLDM*X (FLDMDBX, FLDMIAX) — Increment After	-
1		0	0		01		VSTR — half-precision scalar	FEAT_FP16
1		0	0		10		VSTR — single-precision scalar	-
1		0	0		11		VSTR — double-precision scalar	-

Decode fields							Instruction Details	Feature
P	U	W	L	Rn	size	imm8		
1		0	1	!= 1111	01		VLDR (immediate) — half-precision scalar	FEAT_FP16
1		0	1	!= 1111	10		VLDR (immediate) — single-precision scalar	-
1		0	1	!= 1111	11		VLDR (immediate) — double-precision scalar	-
1	0	1			0x		UNALLOCATED	-
1	0	1	0		10		VSTM, VSTMDB, VSTMIA — single-precision scalar	-
1	0	1	0		11	xxxxxxx0	VSTM, VSTMDB, VSTMIA — double-precision scalar	-
1	0	1	0		11	xxxxxxx1	FSTMDBX, FSTMIAX — Decrement Before	-
1	0	1	1		10		VLDM, VLDMDB, VLDMIA — single-precision scalar	-
1	0	1	1		11	xxxxxxx0	VLDM, VLDMDB, VLDMIA — double-precision scalar	-
1	0	1	1		11	xxxxxxx1	FLDM*X (FLDMDBX, FLDMIAX) — Decrement Before	-
1		0	1	1111	01		VLDR (literal) — half-precision scalar	FEAT_FP16
1		0	1	1111	10		VLDR (literal) — single-precision scalar	-
1		0	1	1111	11		VLDR (literal) — double-precision scalar	-
1	1	1					UNALLOCATED	-

System register load/store

These instructions are under Advanced SIMD and System register load/store and 64-bit move.

31 30 29 28	27	26	25	24	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	L	Rn	CRd	1	1	1	cp15	imm8
cond															

The following constraints also apply to this encoding: cond != 1111 && P:U:D:W != 00x0 && cond != 1111

Decode fields						Instruction Details
P:U:W	D	L	Rn	CRd	cp15	
!= 000	0			!= 0101	0	UNALLOCATED
!= 000	0	1	1111	0101	0	LDC (literal)
!= 000					1	UNALLOCATED
!= 000	1			0101	0	UNALLOCATED
0x1	0	0		0101	0	STC — post-indexed
0x1	0	1	!= 1111	0101	0	LDC (immediate) — post-indexed
010	0	0		0101	0	STC — unindexed
010	0	1	!= 1111	0101	0	LDC (immediate) — unindexed
1x0	0	0		0101	0	STC — offset
1x0	0	1	!= 1111	0101	0	LDC (immediate) — offset
1x1	0	0		0101	0	STC — pre-indexed
1x1	0	1	!= 1111	0101	0	LDC (immediate) — pre-indexed

Advanced SIMD and System register 32-bit move

These instructions are under System register access, Advanced SIMD, floating-point, and Supervisor call.

31-28	27-24	23-21	20-12	11	10-8	7-5	4	3-0
!= 1111	1110	op0		1	op1		1	

Decode fields op0	op1	Instruction details	Feature
000	000	UNALLOCATED	-
000	001	VMOV (between general-purpose register and half-precision)	FEAT_FP16
000	010	VMOV (between general-purpose register and single-precision)	-
001	010	UNALLOCATED	-
01x	010	UNALLOCATED	-
10x	010	UNALLOCATED	-
110	010	UNALLOCATED	-
111	010	Floating-point move special register	-
	011	Advanced SIMD 8/16/32-bit element move/duplicate	-
	10x	UNALLOCATED	-
	11x	System register 32-bit move	-

Floating-point move special register

These instructions are under Advanced SIMD and System register 32-bit move.

31-28	27	26	25	24	23	22	21	20	19-16	15-12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111	1	1	1	0	1	1	1	L	reg	Rt	1	0	1	0	(0)	(0)	(0)	1	(0)	(0)	(0)	(0)
cond																						

The following constraints also apply to this encoding: cond != 1111 && cond != 1111

Decode fields L	Instruction Details
0	VMSR
1	VMRS

Advanced SIMD 8/16/32-bit element move/duplicate

These instructions are under Advanced SIMD and System register 32-bit move.

31-28	27	26	25	24	23-21	20	19-16	15-12	11	10	9	8	7	6-5	4	3	2	1	0
!= 1111	1	1	1	0	opc1	L	Vn	Rt	1	0	1	1	N	opc2	1	(0)	(0)	(0)	(0)
cond																			

The following constraints also apply to this encoding: cond != 1111 && cond != 1111

Decode fields opc1	L	opc2	Instruction Details
0xx	0		VMOV (general-purpose register to scalar)
	1		VMOV (scalar to general-purpose register)
1xx	0	0x	VDUP (general-purpose register)
1xx	0	1x	UNALLOCATED

System register 32-bit move

These instructions are under Advanced SIMD and System register 32-bit move.

31-28	27	26	25	24	23-21	20	19-16	15-12	11	10	9	8	7-5	4	3-0
!= 1111	1	1	1	0	opc1	L	CRn	Rt	1	1	1	cp15	opc2	1	CRm
cond															

The following constraints also apply to this encoding: cond != 1111 && cond != 1111

Decode fields L	Instruction Details
0	MCR
1	MRC

Floating-point data-processing

These instructions are under System register access, Advanced SIMD, floating-point, and Supervisor call.

31–28	27–24	23–20	19–12	11–10	9–7	6	5	4	3–0
!= 1111	1110	op0		10		op1		0	

Decode fields op0	op1	Instruction details
1x11	1	Floating-point data-processing (two registers)
1x11	0	Floating-point move immediate
!= 1x11		Floating-point data-processing (three registers)

Floating-point data-processing (two registers)

These instructions are under Floating-point data-processing.

31–28	27	26	25	24	23	22	21	20	19	18–16	15–12	11	10	9–8	7	6	5	4	3–0
!= 1111	1	1	1	0	1	D	1	1	o1	opc2	Vd	1	0	size	o3	1	M	0	Vm
cond																			

The following constraints also apply to this encoding: cond != 1111 && cond != 1111

o1	opc2	size	o3	Instruction Details	Feature
		00		UNALLOCATED	-
0	000	01	0	UNALLOCATED	-
0	000	01	1	VABS — half-precision scalar	FEAT_FP16
0	000	10	0	VMOV (register) — single-precision scalar	-
0	000	10	1	VABS — single-precision scalar	-
0	000	11	0	VMOV (register) — double-precision scalar	-
0	000	11	1	VABS — double-precision scalar	-
0	001	01	0	VNEG — half-precision scalar	FEAT_FP16
0	001	01	1	VSQRT — half-precision scalar	FEAT_FP16
0	001	10	0	VNEG — single-precision scalar	-
0	001	10	1	VSQRT — single-precision scalar	-
0	001	11	0	VNEG — double-precision scalar	-
0	001	11	1	VSQRT — double-precision scalar	-
0	010	01		UNALLOCATED	-
0	010	10	0	VCVTB — half-precision to single-precision	-
0	010	10	1	VCVTT — half-precision to single-precision	-
0	010	11	0	VCVTB — half-precision to double-precision	-
0	010	11	1	VCVTT — half-precision to double-precision	-
0	011	01	0	VCVTB (BFloat16)	FEAT_AA32BF16
0	011	01	1	VCVTT (BFloat16)	FEAT_AA32BF16
0	011	10	0	VCVTB — single-precision to half-precision	-

Decode fields o1	opc2	size	o3	Instruction Details	Feature
0	011	10	1	VCVTT — single-precision to half-precision	-
0	011	11	0	VCVTB — double-precision to half-precision	-
0	011	11	1	VCVTT — double-precision to half-precision	-
0	100	01	0	VCMP	FEAT_FP16
0	100	01	1	VCMPE	FEAT_FP16
0	100	10	0	VCMP	-
0	100	10	1	VCMPE	-
0	100	11	0	VCMP	-
0	100	11	1	VCMPE	-
0	101	01	0	VCMP	FEAT_FP16
0	101	01	1	VCMPE	FEAT_FP16
0	101	10	0	VCMP	-
0	101	10	1	VCMPE	-
0	101	11	0	VCMP	-
0	101	11	1	VCMPE	-
0	110	01	0	VRINTR — half-precision scalar	FEAT_FP16
0	110	01	1	VRINTZ (floating-point) — half-precision scalar	FEAT_FP16
0	110	10	0	VRINTR — single-precision scalar	-
0	110	10	1	VRINTZ (floating-point) — single-precision scalar	-
0	110	11	0	VRINTR — double-precision scalar	-
0	110	11	1	VRINTZ (floating-point) — double-precision scalar	-
0	111	01	0	VRINTX (floating-point) — half-precision scalar	FEAT_FP16
0	111	01	1	UNALLOCATED	-
0	111	10	0	VRINTX (floating-point) — single-precision scalar	-
0	111	10	1	VCVT (between double-precision and single-precision) — single-precision to double-precision	-
0	111	11	0	VRINTX (floating-point) — double-precision scalar	-
0	111	11	1	VCVT (between double-precision and single-precision) — double-precision to single-precision	-
1	000	01		VCVT (integer to floating-point, floating-point) — half-precision scalar	FEAT_FP16
1	000	10		VCVT (integer to floating-point, floating-point) — single-precision scalar	-
1	000	11		VCVT (integer to floating-point, floating-point) — double-precision scalar	-
1	001	01		UNALLOCATED	-
1	001	10		UNALLOCATED	-
1	001	11	0	UNALLOCATED	-
1	001	11	1	VJCVT	FEAT_JSCVT
1	01x	01		VCVT (between floating-point and fixed-point, floating-point)	FEAT_FP16
1	01x	10		VCVT (between floating-point and fixed-point, floating-point)	-
1	01x	11		VCVT (between floating-point and fixed-point, floating-point)	-
1	100	01	0	VCVTR	FEAT_FP16
1	100	01	1	VCVT (floating-point to integer, floating-point)	FEAT_FP16
1	100	10	0	VCVTR	-
1	100	10	1	VCVT (floating-point to integer, floating-point)	-
1	100	11	0	VCVTR	-
1	100	11	1	VCVT (floating-point to integer, floating-point)	-
1	101	01	0	VCVTR	FEAT_FP16

Decode fields o1	opc2	size	o3	Instruction Details	Feature
1	101	01	1	VCVT (floating-point to integer, floating-point)	FEAT_FP16
1	101	10	0	VCVTR	-
1	101	10	1	VCVT (floating-point to integer, floating-point)	-
1	101	11	0	VCVTR	-
1	101	11	1	VCVT (floating-point to integer, floating-point)	-
1	11x	01		VCVT (between floating-point and fixed-point, floating-point)	FEAT_FP16
1	11x	10		VCVT (between floating-point and fixed-point, floating-point)	-
1	11x	11		VCVT (between floating-point and fixed-point, floating-point)	-

Floating-point move immediate

These instructions are under Floating-point data-processing.

31 30 29 28	27	26	25	24	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	imm4H	Vd	1	0	size	(0)	0	(0)	0	imm4L
cond																		

The following constraints also apply to this encoding: cond != 1111 && cond != 1111

Decode fields size	Instruction Details	Feature
00	UNALLOCATED	-
01	VMOV (immediate) — half-precision scalar	FEAT_FP16
10	VMOV (immediate) — single-precision scalar	-
11	VMOV (immediate) — double-precision scalar	-

Floating-point data-processing (three registers)

These instructions are under Floating-point data-processing.

31 30 29 28	27	26	25	24	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	o0	D	o1	Vn	Vd	1	0	size	N	o2	M	0	Vm
cond																	

The following constraints also apply to this encoding: cond != 1111 && o0:D:o1 != 1x11 && cond != 1111

Decode fields o0:o1	size	o2	Instruction Details	Feature
!= 111	00		UNALLOCATED	-
000	01	0	VMLA (floating-point) — half-precision scalar	FEAT_FP16
000	01	1	VMLS (floating-point) — half-precision scalar	FEAT_FP16
000	10	0	VMLA (floating-point) — single-precision scalar	-
000	10	1	VMLS (floating-point) — single-precision scalar	-
000	11	0	VMLA (floating-point) — double-precision scalar	-
000	11	1	VMLS (floating-point) — double-precision scalar	-
001	01	0	VNMLS — half-precision scalar	FEAT_FP16
001	01	1	VNMLA — half-precision scalar	FEAT_FP16
001	10	0	VNMLS — single-precision scalar	-
001	10	1	VNMLA — single-precision scalar	-
001	11	0	VNMLS — double-precision scalar	-
001	11	1	VNMLA — double-precision scalar	-

Decode fields o0:o1	size	o2	Instruction Details	Feature
010	01	0	VMUL (floating-point) — half-precision scalar	FEAT_FP16
010	01	1	VNMUL — half-precision scalar	FEAT_FP16
010	10	0	VMUL (floating-point) — single-precision scalar	-
010	10	1	VNMUL — single-precision scalar	-
010	11	0	VMUL (floating-point) — double-precision scalar	-
010	11	1	VNMUL — double-precision scalar	-
011	01	0	VADD (floating-point) — half-precision scalar	FEAT_FP16
011	01	1	VSUB (floating-point) — half-precision scalar	FEAT_FP16
011	10	0	VADD (floating-point) — single-precision scalar	-
011	10	1	VSUB (floating-point) — single-precision scalar	-
011	11	0	VADD (floating-point) — double-precision scalar	-
011	11	1	VSUB (floating-point) — double-precision scalar	-
100	01	0	VDIV — half-precision scalar	FEAT_FP16
100	10	0	VDIV — single-precision scalar	-
100	11	0	VDIV — double-precision scalar	-
101	01	0	VFNMS — half-precision scalar	FEAT_FP16
101	01	1	VFNMA — half-precision scalar	FEAT_FP16
101	10	0	VFNMS — single-precision scalar	-
101	10	1	VFNMA — single-precision scalar	-
101	11	0	VFNMS — double-precision scalar	-
101	11	1	VFNMA — double-precision scalar	-
110	01	0	VFMA — half-precision scalar	FEAT_FP16
110	01	1	VFMS — half-precision scalar	FEAT_FP16
110	10	0	VFMA — single-precision scalar	-
110	10	1	VFMS — single-precision scalar	-
110	11	0	VFMA — double-precision scalar	-
110	11	1	VFMS — double-precision scalar	-

Unconditional 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
11110	op0		op1	

Decode fields op0	op1	Instruction details
00x		Miscellaneous
01x		Advanced SIMD data-processing
1xx	1	Memory hints and barriers
100	0	Advanced SIMD element or structure load/store
101	0	UNALLOCATED
11x	0	UNALLOCATED

Miscellaneous

These instructions are under Unconditional instructions.

31–25	24–20	19–8	7–4	3–0
1111000	op0		op1	

Decode fields op0	op1	Instruction details	Feature
0xxxx		UNALLOCATED	-
10000	xx0x	Change Process State	-
10001	1000	UNALLOCATED	-
10001	x100	UNALLOCATED	-
10001	xx01	UNALLOCATED	-
10001	0000	SETPAN	FEAT_PAN
1000x	0111	UNALLOCATED	-
10010	0111	CONSTRAINED UNPREDICTABLE	-
10011	0111	UNALLOCATED	-
1001x	xx0x	UNALLOCATED	-
100xx	0011	UNALLOCATED	-
100xx	0x10	UNALLOCATED	-
100xx	1x1x	UNALLOCATED	-
101xx		UNALLOCATED	-
11xxx		UNALLOCATED	-

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.

31	30	29	28	27	26	25	24	23	22	21	20	19–18	17	16	15	14	13	12	11	10	9	8	7	6	5	4–0
1	1	1	1	0	0	0	1	0	0	0	0	imod	M	op	(0)	(0)	(0)	(0)	(0)	(0)	E	A	I	F	0	mode

Decode fields imod	M	op	I	F	mode	Instruction Details
		1	0	0	0xxxx	SETEND
00	1	0				CPS, CPSID, CPSIE — change mode
10		0				CPS, CPSID, CPSIE — interrupt enable and change mode
		1	0	0	1xxxx	UNALLOCATED
		1	0	1		UNALLOCATED
		1	1			UNALLOCATED
11		0				CPS, CPSID, CPSIE — interrupt disable and change mode

Advanced SIMD data-processing

These instructions are under Unconditional instructions.

31–25	24	23	22–5	4	3–0
1111001		op0		op1	

Decode fields op0	op1	Instruction details
0		Advanced SIMD three registers of the same length
1	0	Advanced SIMD two registers, or three registers of different lengths
1	1	Advanced SIMD shifts and immediate generation

Advanced SIMD three registers of the same length

These instructions are under Advanced SIMD data-processing.

31	30	29	28	27	26	25	24	23	22	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	opc	N	Q	M	o1	Vm

Decode fields U	size	opc	Q	o1	Instruction Details	Feature
0	0x	1100		1	VFMA	-
0	0x	1101		0	VADD (floating-point)	-
0	0x	1101		1	VMLA (floating-point)	-
0	0x	1110		0	VCEQ (register) — A2	-
0	0x	1111		0	VMAX (floating-point)	-
0	0x	1111		1	VRECPS	-
		0000		0	VHADD	-
0	00	0001		1	VAND (register)	-
		0000		1	VQADD	-
		0001		0	VRHADD	-
0	00	1100		0	SHA1C	FEAT_SHA1
		0010		0	VHSUB	-
0	01	0001		1	VBIC (register)	-
		0010		1	VQSUB	-
		0011		0	VCGT (register) — A1	-
		0011		1	VCGE (register) — A1	-
0	01	1100		0	SHA1P	FEAT_SHA1
0	1x	1100		1	VFMS	-
0	1x	1101		0	VSUB (floating-point)	-
0	1x	1101		1	VMLS (floating-point)	-
0	1x	1110		0	UNALLOCATED	-
0	1x	1111		0	VMIN (floating-point)	-
0	1x	1111		1	VRSQRTS	-
		0100		0	VSHL (register)	-
0		1000		0	VADD (integer)	-
0	10	0001		1	VORR (register)	-
0		1000		1	VTST	-
		0100		1	VQSHL (register)	-
0		1001		0	VMLA (integer)	-
		0101		0	VRSHL	-
		0101		1	VQRSHL	-
0		1011		0	VQDMULH	-
0	10	1100		0	SHA1M	FEAT_SHA1
0		1011		1	VPADD (integer)	-
		0110		0	VMAX (integer)	-
0	11	0001		1	VORN (register)	-
		0110		1	VMIN (integer)	-
		0111		0	VABD (integer)	-
		0111		1	VABA	-
0	11	1100		0	SHA1SU0	FEAT_SHA1
1	0x	1101		0	VPADD (floating-point)	-
1	0x	1101		1	VMUL (floating-point)	-

Decode fields U	size	opc	Q	o1	Instruction Details	Feature
1	0x	1110		0	VCGE (register) — A2	-
1	0x	1110		1	VACGE	-
1	0x	1111	0	0	VPMAX (floating-point)	-
1	0x	1111		1	VMAXNM	-
1	00	0001		1	VEOR	-
		1001		1	VMUL (integer and polynomial)	-
1	00	1100		0	SHA256H	FEAT_SHA256
		1010	0	0	VPMAX (integer)	-
1	01	0001		1	VBSL	-
		1010	0	1	VPMIN (integer)	-
		1010	1		UNALLOCATED	-
1	01	1100		0	SHA256H2	FEAT_SHA256
1	1x	1101		0	VABD (floating-point)	-
1	1x	1110		0	VCGT (register) — A2	-
1	1x	1110		1	VACGT	-
1	1x	1111	0	0	VPMIN (floating-point)	-
1	1x	1111		1	VMINNM	-
1		1000		0	VSUB (integer)	-
1	10	0001		1	VBIT	-
1		1000		1	VCEQ (register) — A1	-
1		1001		0	VMLS (integer)	-
1		1011		0	VQRDMULH	-
1	10	1100		0	SHA256SU1	FEAT_SHA256
1		1011		1	VQRDMLAH	FEAT_RDM
1	11	0001		1	VBIF	-
1		1100		1	VQRDMLSH	FEAT_RDM
1		1111	1	0	UNALLOCATED	-

Advanced SIMD two registers, or three registers of different lengths

These instructions are under Advanced SIMD data-processing.

31–25	24	23	22	21–20	19–12	11–10	9–7	6	5	4	3–0
1111001	op0	1		op1		op2		op3		0	

Decode fields op0	op1	op2	op3	Instruction details
0	11			VEXT (byte elements)
1	11	0x		Advanced SIMD two registers misc
1	11	10		VTBL, VTBX
1	11	11		Advanced SIMD duplicate (scalar)
	!= 11		0	Advanced SIMD three registers of different lengths
	!= 11		1	Advanced SIMD two registers and a scalar

Advanced SIMD two registers misc

These instructions are under Advanced SIMD two registers, or three registers of different lengths.

31	30	29	28	27	26	25	24	23	22	21	20	19–18	17–16	15–12	11	10–7	6	5	4	3–0
1	1	1	1	0	0	1	1	1	D	1	1	size	opc1	Vd	0	opc2	Q	M	0	Vm

size	Decode fields opc1	opc2	Q	Instruction Details	Feature
	00	0000		VREV64	-
	00	0001		VREV32	-
	00	0010		VREV16	-
	00	0011		UNALLOCATED	-
	00	010x		VPADDL	-
	00	0110	0	AESE	FEAT_AES
	00	0110	1	AESD	FEAT_AES
	00	0111	0	AESMC	FEAT_AES
	00	0111	1	AESIMC	FEAT_AES
	00	1000		VCLS	-
00	10	0000		VSWP	-
	00	1001		VCLZ	-
	00	1010		VCNT	-
	00	1011		VMVN (register)	-
00	10	1100	1	UNALLOCATED	-
	00	110x		VPADAL	-
	00	1110		VQABS	-
	00	1111		VQNEG	-
	01	x000		VCGT (immediate #0)	-
	01	x001		VCGE (immediate #0)	-
	01	x010		VCEQ (immediate #0)	-
	01	x011		VCLE (immediate #0)	-
	01	x100		VCLT (immediate #0)	-
	01	x110		VABS	-
	01	x111		VNEG	-
	01	0101	1	SHA1H	FEAT_SHA1
01	10	1100	1	VCVT (from single-precision to BFloat16, Advanced SIMD)	FEAT_AA32BF16
	10	0001		VTRN	-
	10	0010		VUZP	-
	10	0011		VZIP	-
	10	0100	0	VMOVN	-
	10	0100	1	VQMOVN, VQMOVUN — VQMOVUN	-
	10	0101		VQMOVN, VQMOVUN — VQMOVN	-
	10	0110	0	VSHLL	-
	10	0111	0	SHA1SU1	FEAT_SHA1
	10	0111	1	SHA256SU0	FEAT_SHA256
	10	1000		VRINTN (Advanced SIMD)	-
	10	1001		VRINTX (Advanced SIMD)	-
	10	1010		VRINTA (Advanced SIMD)	-
	10	1011		VRINTZ (Advanced SIMD)	-
10	10	1100	1	UNALLOCATED	-
	10	1100	0	VCVT (between half-precision and single-precision, Advanced SIMD) — single-precision to half-precision	-
	10	1101		VRINTM (Advanced SIMD)	-
	10	1110	0	VCVT (between half-precision and single-precision, Advanced SIMD) — half-precision to single-precision	-
	10	1110	1	UNALLOCATED	-
	10	1111		VRINTP (Advanced SIMD)	-

Decode fields size	opc1	opc2	Q	Instruction Details	Feature
	11	000x		VCVTA (Advanced SIMD)	-
	11	001x		VCVTN (Advanced SIMD)	-
	11	010x		VCVTP (Advanced SIMD)	-
	11	011x		VCVTM (Advanced SIMD)	-
	11	10x0		VRECPE	-
	11	10x1		VRSQRTE	-
11	10	1100	1	UNALLOCATED	-
	11	11xx		VCVT (between floating-point and integer, Advanced SIMD)	-

Advanced SIMD duplicate (scalar)

These instructions are under Advanced SIMD two registers, or three registers of different lengths.

31	30	29	28	27	26	25	24	23	22	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	imm4				Vd				1	1	opc			Q	M	0	Vm			

Decode fields opc	Instruction Details
000	VDUP (scalar)
001	UNALLOCATED
01x	UNALLOCATED
1xx	UNALLOCATED

Advanced SIMD three registers of different lengths

These instructions are under Advanced SIMD two registers, or three registers of different lengths.

31	30	29	28	27	26	25	24	23	22	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				opc				N	0	M	0	Vm			
										size																					

The following constraints also apply to this encoding: size != 11 && size != 11

Decode fields U	opc	Instruction Details
	0000	VADDL
	0001	VADDW
	0010	VSUBL
0	0100	VADDHN
	0011	VSUBW
0	0110	VSUBHN
0	1001	VQDMLAL
	0101	VABAL
0	1011	VQDMLSL
0	1101	VQDMULL
	0111	VABDL (integer)
	1000	VMLAL (integer)
	1010	VMLSL (integer)
1	0100	VRADDHN
1	0110	VRSUBHN
	11x0	VMULL (integer and polynomial)

Decode fields U	opc	Instruction Details
1	1001	UNALLOCATED
1	1011	UNALLOCATED
1	1101	UNALLOCATED
	1111	UNALLOCATED

Advanced SIMD two registers and a scalar

These instructions are under Advanced SIMD two registers, or three registers of different lengths.

31	30	29	28	27	26	25	24	23	22	21-20	19-16	15-12	11-8	7	6	5	4	3-0
1	1	1	1	0	0	1	Q	1	D	!= 11	Vn	Vd	opc	N	1	M	0	Vm
										size								

The following constraints also apply to this encoding: size != 11 && size != 11

Decode fields Q	opc	Instruction Details	Feature
	000x	VMLA (by scalar)	-
0	0011	VQDMLAL	-
	0010	VMLAL (by scalar)	-
0	0111	VQDMLSL	-
	010x	VMLS (by scalar)	-
0	1011	VQDMULL	-
	0110	VMLSL (by scalar)	-
	100x	VMUL (by scalar)	-
1	0011	UNALLOCATED	-
	1010	VMULL (by scalar)	-
1	0111	UNALLOCATED	-
	1100	VQDMULH	-
	1101	VQRDMULH	-
1	1011	UNALLOCATED	-
	1110	VQRDMLAH	FEAT_RDM
	1111	VQRDMLSH	FEAT_RDM

Advanced SIMD shifts and immediate generation

These instructions are under Advanced SIMD data-processing.

31-25	24	23	22	21-7	6-5	4	3-0
1111001		1		op0		1	

Decode fields op0	Instruction details
000xxxxxxxxxxx0	Advanced SIMD one register and modified immediate
!= 000xxxxxxxxxxx0	Advanced SIMD two registers and shift amount

Advanced SIMD one register and modified immediate

These instructions are under Advanced SIMD shifts and immediate generation.

31	30	29	28	27	26	25	24	23	22	21	20	19	18-16	15-12	11-8	7	6	5	4	3-0
1	1	1	1	0	0	1	i	1	D	0	0	0	imm3	Vd	cmode	0	Q	op	1	imm4

Decode fields cmode	op	Instruction Details
0xx0	0	VMOV (immediate) — A1
0xx0	1	VMVN (immediate) — A1
0xx1	0	VORR (immediate) — A1
0xx1	1	VBIC (immediate) — A1
10x0	0	VMOV (immediate) — A3
10x0	1	VMVN (immediate) — A2
10x1	0	VORR (immediate) — A2
10x1	1	VBIC (immediate) — A2
11xx	0	VMOV (immediate) — A4
110x	1	VMVN (immediate) — A3
1110	1	VMOV (immediate) — A5
1111	1	UNALLOCATED

Advanced SIMD two registers and shift amount

These instructions are under Advanced SIMD shifts and immediate generation.

31	30	29	28	27	26	25	24	23	22	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	imm3H	imm3L	Vd	opc	L	Q	M	1	Vm

The following constraints also apply to this encoding: imm3H:imm3L:Vd:opc:L != 000xxxxxxxxxxx0

Decode fields U	imm3H:L	imm3L	opc	Q	Instruction Details
	!= 0000		0000		VSHR
	!= 0000		0001		VSRA
	!= 0000	000	1010	0	VMOVL
	!= 0000		0010		VRSHR
	!= 0000		0011		VRSRA
	!= 0000		0111		VQSHL, VQSHLU (immediate) — VQSHL
	!= 0000		1001	0	VQSHRN, VQSHRUN — VQSHRN
	!= 0000		1001	1	VQRSHRN, VQRSHRUN — VQRSHRN
	!= 0000		1010	0	VSHLL
	!= 0000		11xx		VCVT (between floating-point and fixed-point, Advanced SIMD)
0	!= 0000		0101		VSHL (immediate)
0	!= 0000		1000	0	VSHRN
0	!= 0000		1000	1	VRSHRN
1	!= 0000		0100		VSRI
1	!= 0000		0101		VSLI
1	!= 0000		0110		VQSHL, VQSHLU (immediate) — VQSHLU
1	!= 0000		1000	0	VQSHRN, VQSHRUN — VQSHRUN
1	!= 0000		1000	1	VQRSHRN, VQRSHRUN — VQRSHRUN

Memory hints and barriers

These instructions are under Unconditional instructions.

31 30 29 28 27 26	25 24 23 22 21	20	19 18 17 16 15 14 13 12 11 10 9 8 7 6 5	4	3 2 1 0
111101	op0	1		op1	

Decode fields op0	op1	Instruction details
00xx1		CONSTRAINED UNPREDICTABLE
01001		CONSTRAINED UNPREDICTABLE
01011		Barriers
011x1		CONSTRAINED UNPREDICTABLE
0xxx0		Preload (immediate)
1xxx0	0	Preload (register)
1xxx1	0	CONSTRAINED UNPREDICTABLE
1xxxx	1	UNALLOCATED

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.

31	30	29	28	27	26	25	24	23	22	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	0	1	1	1	(1)	(1)	(1)	(1)	(1)	(1)	(1)	(1)	(0)	(0)	(0)	(0)	opcode				option			

Decode fields opcode	option	Instruction Details	Feature
0000		CONSTRAINED UNPREDICTABLE	-
0001		CLREX	-
001x		CONSTRAINED UNPREDICTABLE	-
0100	!= 0x00	DSB	-
0100	0000	SSBB	-
0100	0100	PSSBB	-
0101		DMB	-
0110		ISB	-
0111		SB	FEAT_SB
1xxx		CONSTRAINED UNPREDICTABLE	-

The behavior of the CONSTRAINED UNPREDICTABLE encodings in this table is described in *CONSTRAINED UNPREDICTABLE behavior for A32 and T32 instruction encodings*

Preload (immediate)

These instructions are under Memory hints and barriers.

31	30	29	28	27	26	25	24	23	22	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	U	R	0	1	Rn				(1)	(1)	(1)	(1)	imm12											

Decode fields D	R	Rn	Instruction Details
0	0		Reserved hint, behaves as NOP
0	1		PLI (immediate, literal)
1		1111	PLD (literal)
1	0	!= 1111	PLD, PLDW (immediate) — preload write
1	1	!= 1111	PLD, PLDW (immediate) — preload read

Preload (register)

These instructions are under Memory hints and barriers.

31	30	29	28	27	26	25	24	23	22	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	D	U	o2	0	1	Rn	(1)	(1)	(1)	(1)	imm5	stype	0	Rm

Decode fields D	o2	imm5:stype	Instruction Details
0	0		Reserved hint, behaves as NOP
0	1	!= 0000011	PLI (register) — shift or rotate by value
0	1	0000011	PLI (register) — rotate right with extend
1	0	!= 0000011	PLD, PLDW (register) — preload write, optional shift or rotate
1	0	0000011	PLD, PLDW (register) — preload write, rotate right with extend
1	1	!= 0000011	PLD, PLDW (register) — preload read, optional shift or rotate
1	1	0000011	PLD, PLDW (register) — preload read, rotate right with extend

Advanced SIMD element or structure load/store

These instructions are under Unconditional instructions.

31 30 29 28 27 26 25 24	23	22 21	20	19 18 17 16 15 14 13 12	11 10	9 8 7 6 5 4 3 2 1 0
11110100	op0		0		op1	

Decode fields op0	op1	Instruction details
0		Advanced SIMD load/store multiple structures
1	11	Advanced SIMD load single structure to all lanes
1	!= 11	Advanced SIMD load/store single structure to one lane

Advanced SIMD load/store multiple structures

These instructions are under Advanced SIMD element or structure load/store.

31	30	29	28	27	26	25	24	23	22	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	L	0	Rn	Vd	itype	size	align	Rm

Decode fields L	itype	Rm	Instruction Details
0	000x	!= 11x1	VST4 (multiple 4-element structures)
0	000x	1101	VST4 (multiple 4-element structures)
0	000x	1111	VST4 (multiple 4-element structures)
0	0010	!= 11x1	VST1 (multiple single elements)
0	0010	1101	VST1 (multiple single elements)
0	0010	1111	VST1 (multiple single elements)
0	0011	!= 11x1	VST2 (multiple 2-element structures)
0	0011	1101	VST2 (multiple 2-element structures)
0	0011	1111	VST2 (multiple 2-element structures)
0	010x	!= 11x1	VST3 (multiple 3-element structures)
0	010x	1101	VST3 (multiple 3-element structures)
0	010x	1111	VST3 (multiple 3-element structures)
0	0110	!= 11x1	VST1 (multiple single elements)
0	0110	1101	VST1 (multiple single elements)
0	0110	1111	VST1 (multiple single elements)

Decode fields			Instruction Details
L	itype	Rm	
0	0111	!= 11x1	VST1 (multiple single elements)
0	0111	1101	VST1 (multiple single elements)
0	0111	1111	VST1 (multiple single elements)
0	100x	!= 11x1	VST2 (multiple 2-element structures)
0	100x	1101	VST2 (multiple 2-element structures)
0	100x	1111	VST2 (multiple 2-element structures)
0	1010	!= 11x1	VST1 (multiple single elements)
0	1010	1101	VST1 (multiple single elements)
0	1010	1111	VST1 (multiple single elements)
1	000x	!= 11x1	VLD4 (multiple 4-element structures)
1	000x	1101	VLD4 (multiple 4-element structures)
1	000x	1111	VLD4 (multiple 4-element structures)
1	0010	!= 11x1	VLD1 (multiple single elements)
1	0010	1101	VLD1 (multiple single elements)
1	0010	1111	VLD1 (multiple single elements)
1	0011	!= 11x1	VLD2 (multiple 2-element structures)
1	0011	1101	VLD2 (multiple 2-element structures)
1	0011	1111	VLD2 (multiple 2-element structures)
1	010x	!= 11x1	VLD3 (multiple 3-element structures)
1	010x	1101	VLD3 (multiple 3-element structures)
1	010x	1111	VLD3 (multiple 3-element structures)
	1011		UNALLOCATED
1	0110	!= 11x1	VLD1 (multiple single elements)
1	0110	1101	VLD1 (multiple single elements)
1	0110	1111	VLD1 (multiple single elements)
1	0111	!= 11x1	VLD1 (multiple single elements)
1	0111	1101	VLD1 (multiple single elements)
1	0111	1111	VLD1 (multiple single elements)
	11xx		UNALLOCATED
1	100x	!= 11x1	VLD2 (multiple 2-element structures)
1	100x	1101	VLD2 (multiple 2-element structures)
1	100x	1111	VLD2 (multiple 2-element structures)
1	1010	!= 11x1	VLD1 (multiple single elements)
1	1010	1101	VLD1 (multiple single elements)
1	1010	1111	VLD1 (multiple single elements)

Advanced SIMD load single structure to all lanes

These instructions are under Advanced SIMD element or structure load/store.

31	30	29	28	27	26	25	24	23	22	21	20	19-16	15-12	11	10	9-8	7-6	5	4	3-0
1	1	1	1	0	1	0	0	1	D	L	0	Rn	Vd	1	1	N	size	T	a	Rm

Decode fields				Instruction Details
L	N	a	Rm	
0				UNALLOCATED
1	00		!= 11x1	VLD1 (single element to all lanes)
1	00		1101	VLD1 (single element to all lanes)
1	00		1111	VLD1 (single element to all lanes)

L	N	a	Rm	Instruction Details
1	01		!= 11x1	VLD2 (single 2-element structure to all lanes)
1	01		1101	VLD2 (single 2-element structure to all lanes)
1	01		1111	VLD2 (single 2-element structure to all lanes)
1	10	0	!= 11x1	VLD3 (single 3-element structure to all lanes)
1	10	0	1101	VLD3 (single 3-element structure to all lanes)
1	10	0	1111	VLD3 (single 3-element structure to all lanes)
1	10	1		UNALLOCATED
1	11		!= 11x1	VLD4 (single 4-element structure to all lanes)
1	11		1101	VLD4 (single 4-element structure to all lanes)
1	11		1111	VLD4 (single 4-element structure to all lanes)

Advanced SIMD load/store single structure to one lane

These instructions are under Advanced SIMD element or structure load/store.

31	30	29	28	27	26	25	24	23	22	21	20	19-16	15-12	11-10	9-8	7-4	3-0
1	1	1	1	0	1	0	0	1	D	L	0	Rn	Vd	!= 11	N	index_align	Rm
														size			

The following constraints also apply to this encoding: size != 11 && size != 11

L	size	N	Rm	Instruction Details
0	00	00	!= 11x1	VST1 (single element from one lane)
0	00	00	1101	VST1 (single element from one lane)
0	00	00	1111	VST1 (single element from one lane)
0	00	01	!= 11x1	VST2 (single 2-element structure from one lane)
0	00	01	1101	VST2 (single 2-element structure from one lane)
0	00	01	1111	VST2 (single 2-element structure from one lane)
0	00	10	!= 11x1	VST3 (single 3-element structure from one lane)
0	00	10	1101	VST3 (single 3-element structure from one lane)
0	00	10	1111	VST3 (single 3-element structure from one lane)
0	00	11	!= 11x1	VST4 (single 4-element structure from one lane)
0	00	11	1101	VST4 (single 4-element structure from one lane)
0	00	11	1111	VST4 (single 4-element structure from one lane)
0	01	00	!= 11x1	VST1 (single element from one lane)
0	01	00	1101	VST1 (single element from one lane)
0	01	00	1111	VST1 (single element from one lane)
0	01	01	!= 11x1	VST2 (single 2-element structure from one lane)
0	01	01	1101	VST2 (single 2-element structure from one lane)
0	01	01	1111	VST2 (single 2-element structure from one lane)
0	01	10	!= 11x1	VST3 (single 3-element structure from one lane)
0	01	10	1101	VST3 (single 3-element structure from one lane)
0	01	10	1111	VST3 (single 3-element structure from one lane)
0	01	11	!= 11x1	VST4 (single 4-element structure from one lane)
0	01	11	1101	VST4 (single 4-element structure from one lane)
0	01	11	1111	VST4 (single 4-element structure from one lane)
0	10	00	!= 11x1	VST1 (single element from one lane)
0	10	00	1101	VST1 (single element from one lane)

Decode fields				Instruction Details
L	size	N	Rm	
0	10	00	1111	VST1 (single element from one lane)
0	10	01	!= 11x1	VST2 (single 2-element structure from one lane)
0	10	01	1101	VST2 (single 2-element structure from one lane)
0	10	01	1111	VST2 (single 2-element structure from one lane)
0	10	10	!= 11x1	VST3 (single 3-element structure from one lane)
0	10	10	1101	VST3 (single 3-element structure from one lane)
0	10	10	1111	VST3 (single 3-element structure from one lane)
0	10	11	!= 11x1	VST4 (single 4-element structure from one lane)
0	10	11	1101	VST4 (single 4-element structure from one lane)
0	10	11	1111	VST4 (single 4-element structure from one lane)
1	00	00	!= 11x1	VLD1 (single element to one lane)
1	00	00	1101	VLD1 (single element to one lane)
1	00	00	1111	VLD1 (single element to one lane)
1	00	01	!= 11x1	VLD2 (single 2-element structure to one lane)
1	00	01	1101	VLD2 (single 2-element structure to one lane)
1	00	01	1111	VLD2 (single 2-element structure to one lane)
1	00	10	!= 11x1	VLD3 (single 3-element structure to one lane)
1	00	10	1101	VLD3 (single 3-element structure to one lane)
1	00	10	1111	VLD3 (single 3-element structure to one lane)
1	00	11	!= 11x1	VLD4 (single 4-element structure to one lane)
1	00	11	1101	VLD4 (single 4-element structure to one lane)
1	00	11	1111	VLD4 (single 4-element structure to one lane)
1	01	00	!= 11x1	VLD1 (single element to one lane)
1	01	00	1101	VLD1 (single element to one lane)
1	01	00	1111	VLD1 (single element to one lane)
1	01	01	!= 11x1	VLD2 (single 2-element structure to one lane)
1	01	01	1101	VLD2 (single 2-element structure to one lane)
1	01	01	1111	VLD2 (single 2-element structure to one lane)
1	01	10	!= 11x1	VLD3 (single 3-element structure to one lane)
1	01	10	1101	VLD3 (single 3-element structure to one lane)
1	01	10	1111	VLD3 (single 3-element structure to one lane)
1	01	11	!= 11x1	VLD4 (single 4-element structure to one lane)
1	01	11	1101	VLD4 (single 4-element structure to one lane)
1	01	11	1111	VLD4 (single 4-element structure to one lane)
1	10	00	!= 11x1	VLD1 (single element to one lane)
1	10	00	1101	VLD1 (single element to one lane)
1	10	00	1111	VLD1 (single element to one lane)
1	10	01	!= 11x1	VLD2 (single 2-element structure to one lane)
1	10	01	1101	VLD2 (single 2-element structure to one lane)
1	10	01	1111	VLD2 (single 2-element structure to one lane)
1	10	10	!= 11x1	VLD3 (single 3-element structure to one lane)
1	10	10	1101	VLD3 (single 3-element structure to one lane)
1	10	10	1111	VLD3 (single 3-element structure to one lane)
1	10	11	!= 11x1	VLD4 (single 4-element structure to one lane)
1	10	11	1101	VLD4 (single 4-element structure to one lane)
1	10	11	1111	VLD4 (single 4-element structure to one lane)

Top-level encodings for T32

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
op0			op1																												

Decode fields op0	op1	Instruction details
!= 111		16-bit
111	00	B — T2
111	!= 00	32-bit

16-bit

These instructions are under the top-level.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
op0															

The following constraints also apply to this encoding: op0<5:3> != 111

Decode fields op0	Instruction details
00xxxx	Shift (immediate), add, subtract, move, and compare
010000	Data-processing (two low registers)
010001	Special data instructions and branch and exchange
01001x	LDR (literal) — T1
0101xx	Load/store (register offset)
011xxx	Load/store word/byte (immediate offset)
1000xx	Load/store halfword (immediate offset)
1001xx	Load/store (SP-relative)
1010xx	Add PC/SP (immediate)
1011xx	Miscellaneous 16-bit instructions
1100xx	Load/store multiple
1101xx	Conditional branch, and Supervisor Call

Shift (immediate), add, subtract, move, and compare

These instructions are under 16-bit.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
00		op0	op1		op2										

Decode fields op0	op1	op2	Instruction details
0	11	0	Add, subtract (three low registers)
0	11	1	Add, subtract (two low registers and immediate)
0	!= 11		MOV, MOVS (register) — T2
1			Add, subtract, compare, move (one low register and immediate)

Add, subtract (three low registers)

These instructions are under Shift (immediate), add, subtract, move, and compare.

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

Decode fields S	Instruction Details
0	ADD, ADDS (register)
1	SUB, SUBS (register)

Add, subtract (two low registers and immediate)

These instructions are under Shift (immediate), add, subtract, move, and compare.

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

Decode fields S	Instruction Details
0	ADD, ADDS (immediate)
1	SUB, SUBS (immediate)

Add, subtract, compare, move (one low register and immediate)

These instructions are under Shift (immediate), add, subtract, move, and compare.

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

Decode fields op	Instruction Details
00	MOV, MOVS (immediate)
01	CMP (immediate)
10	ADD, ADDS (immediate)
11	SUB, SUBS (immediate)

Data-processing (two low registers)

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	0	0	0	op				Rs			Rd		

Decode fields op	Instruction Details
0000	AND, ANDS (register)
0001	EOR, EORS (register)
0010	MOV, MOVS (register-shifted register) — logical shift left
0011	MOV, MOVS (register-shifted register) — logical shift right
0100	MOV, MOVS (register-shifted register) — arithmetic shift right
0101	ADC, ADCS (register)
0110	SBC, SBCS (register)
0111	MOV, MOVS (register-shifted register) — rotate right
1000	TST (register)
1001	RSB, RSBS (immediate)
1010	CMP (register)
1011	CMN (register)

Decode fields op	Instruction Details
1100	ORR, ORRS (register)
1101	MUL, MULS
1110	BIC, BICS (register)
1111	MVN, MVNS (register)

Special data instructions and branch and exchange

These instructions are under 16-bit.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
010001						op0									

Decode fields op0	Instruction details
11	Branch and exchange
!= 11	Add, subtract, compare, move (two high registers)

Branch and exchange

These instructions are under Special data instructions and branch and exchange.

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

Decode fields L	Instruction Details
0	BX
1	BLX (register)

Add, subtract, compare, move (two high registers)

These instructions are under Special data instructions and branch and exchange.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	!= 11		D	Rs				Rd		
						op									

The following constraints also apply to this encoding: op != 11 && op != 11

Decode fields op	D:Rd	Rs	Instruction Details
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		

Decode fields L	B	H	Instruction Details
0	0	0	STR (register)
0	0	1	STRH (register)
0	1	0	STRB (register)
0	1	1	LDRSB (register)
1	0	0	LDR (register)
1	0	1	LDRH (register)
1	1	0	LDRB (register)
1	1	1	LDRSH (register)

Load/store word/byte (immediate 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	1	B	L	imm5					Rn			Rt		

Decode fields B	L	Instruction Details
0	0	STR (immediate)
0	1	LDR (immediate)
1	0	STRB (immediate)
1	1	LDRB (immediate)

Load/store halfword (immediate offset)

These instructions are under 16-bit.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	L	imm5					Rn			Rt		

Decode fields L	Instruction Details
0	STRH (immediate)
1	LDRH (immediate)

Load/store (SP-relative)

These instructions are under 16-bit.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	1	L	Rt			imm8							

Decode fields L	Instruction Details
0	STR (immediate)
1	LDR (immediate)

Add PC/SP (immediate)

These instructions are under 16-bit.

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

Decode fields SP	Instruction Details
0	ADR
1	ADD, ADDS (SP plus immediate)

Miscellaneous 16-bit instructions

These instructions are under 16-bit.

15 14 13 12	11 10 9 8	7	6	5	4 3 2 1 0	
1011	op0	op1	op2		op3	

Decode fields op0	op1	op2	op3	Instruction details	Feature
0000				Adjust SP (immediate)	-
0010				Extend	-
0110	00	0		SETPAN	FEAT_PAN
0110	00	1		UNALLOCATED	-
0110	01			Change Processor State	-
0110	1x			UNALLOCATED	-
0111				UNALLOCATED	-
1000				UNALLOCATED	-
1010	10			HLT	-
1010	!= 10			Reverse bytes	-
1110				BKPT	-
1111			0000	Hints	-
1111			!= 0000	IT	-
x0x1				CBNZ, CBZ	-
x10x				Push and Pop	-

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

Decode fields S	Instruction Details
0	ADD, ADDS (SP plus immediate)
1	SUB, SUBS (SP minus immediate)

Extend

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	1	0	U	B	Rm	Rd

Decode fields U	B	Instruction Details
0	0	SXTH
0	1	SXTB
1	0	UXTH

Decode fields U	B	Instruction Details
1	1	UXTB

Change Processor State

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	1	1	0	0	1	op	flags				

Decode fields op	flags	Instruction Details
0		SETEND
1	0xxxx	CPS, CPSID, CPSIE — interrupt enable
1	1xxxx	CPS, CPSID, CPSIE — interrupt disable

Reverse bytes

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	1	0	1	0	!= 10 (op)		Rm			Rd		

The following constraints also apply to this encoding: op != 10 && op != 10

Decode fields op	Instruction Details
00	REV
01	REV16
11	REVSH

Hints

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	1	1	1	1	hint				0	0	0	0

Decode fields hint	Instruction Details
0000	NOP
0001	YIELD
0010	WFE
0011	WFI
0100	SEV
0101	SEVL
011x	Reserved hint, behaves as NOP
1xxx	Reserved hint, behaves as NOP

Push and Pop

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	L	1	0	P	register_list							

Decode fields L	Instruction Details
0	PUSH
1	POP

Load/store multiple

These instructions are under 16-bit.

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

Decode fields L	Instruction Details
0	STM, STMIA, STMEA
1	LDM, LDMIA, LDMFD

Conditional branch, and Supervisor Call

These instructions are under 16-bit.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1101				op0											

Decode fields op0	Instruction details
111x	Exception generation
!= 111x	B — T1

Exception generation

These instructions are under Conditional branch, and Supervisor Call.

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

Decode fields S	Instruction Details
0	UDF
1	SVC

32-bit

These instructions are under the top-level.

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
111			op0				op1									op3															

The following constraints also apply to this encoding: op0<3:2> != 00

Decode fields op0	op1	op3	Instruction details
x11x			System register access, Advanced SIMD, and floating-point

0100	xx0xx		Load/store multiple
0100	xx1xx		Load/store dual, load/store exclusive, load-acquire/store-release, and table branch
0101			Data-processing (shifted register)
10xx		1	Branches and miscellaneous control
10x0		0	Data-processing (modified immediate)
10x1	xxxx0	0	Data-processing (plain binary immediate)
10x1	xxxx1	0	UNALLOCATED
1100	1xxx0		Advanced SIMD element or structure load/store
1100	!= 1xxx0		Load/store single
1101	0xxxx		Data-processing (register)
1101	10xxx		Multiply, multiply accumulate, and absolute difference
1101	11xxx		Long multiply and divide

System register access, Advanced SIMD, and floating-point

These instructions are under 32-bit.

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
111	op0	11	op1		op2		op3	

Decode fields op0	op1	op2	op3	Instruction details
	0x	0x		UNALLOCATED
	10	0x		UNALLOCATED
	11			Advanced SIMD data-processing
0	0x	1x		Advanced SIMD and System register load/store and 64-bit move
0	10	1x	1	Advanced SIMD and System register 32-bit move
0	10	10	0	Floating-point data-processing
0	10	11	0	UNALLOCATED
1	!= 11	1x		Additional Advanced SIMD and floating-point instructions

Advanced SIMD data-processing

These instructions are under System register access, Advanced SIMD, and floating-point.

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
111		1111	op0		op1	

Decode fields op0	op1	Instruction details
0		Advanced SIMD three registers of the same length
1	0	Advanced SIMD two registers, or three registers of different lengths
1	1	Advanced SIMD shifts and immediate generation

Advanced SIMD three registers of the same length

These instructions are under Advanced SIMD data-processing.

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	opc	N	Q	M	o1	Vm

Decode fields					Instruction Details	Feature
U	size	opc	Q	o1		
0	0x	1100		1	VFMA	-
0	0x	1101		0	VADD (floating-point)	-
0	0x	1101		1	VMLA (floating-point)	-
0	0x	1110		0	VCEQ (register) — T2	-
0	0x	1111		0	VMAX (floating-point)	-
0	0x	1111		1	VRECPS	-
		0000		0	VHADD	-
0	00	0001		1	VAND (register)	-
		0000		1	VQADD	-
		0001		0	VRHADD	-
0	00	1100		0	SHA1C	FEAT_SHA1
		0010		0	VHSUB	-
0	01	0001		1	VBIC (register)	-
		0010		1	VQSUB	-
		0011		0	VCGT (register) — T1	-
		0011		1	VCGE (register) — T1	-
0	01	1100		0	SHA1P	FEAT_SHA1
0	1x	1100		1	VFMS	-
0	1x	1101		0	VSUB (floating-point)	-
0	1x	1101		1	VMLS (floating-point)	-
0	1x	1110		0	UNALLOCATED	-
0	1x	1111		0	VMIN (floating-point)	-
0	1x	1111		1	VRSQRTS	-
		0100		0	VSHL (register)	-
0		1000		0	VADD (integer)	-
0	10	0001		1	VORR (register)	-
0		1000		1	VTST	-
		0100		1	VQSHL (register)	-
0		1001		0	VMLA (integer)	-
		0101		0	VRSHL	-
		0101		1	VQRSHL	-
0		1011		0	VQDMULH	-
0	10	1100		0	SHA1M	FEAT_SHA1
0		1011		1	VPADD (integer)	-
		0110		0	VMAX (integer)	-
0	11	0001		1	VORN (register)	-
		0110		1	VMIN (integer)	-
		0111		0	VABD (integer)	-
		0111		1	VABA	-
0	11	1100		0	SHA1SU0	FEAT_SHA1
1	0x	1101		0	VPADD (floating-point)	-
1	0x	1101		1	VMUL (floating-point)	-
1	0x	1110		0	VCGE (register) — T2	-
1	0x	1110		1	VACGE	-
1	0x	1111	0	0	VPMAX (floating-point)	-
1	0x	1111		1	VMAXNM	-
1	00	0001		1	VEOR	-

U	size	opc	Q	o1	Instruction Details	Feature
		1001		1	VMUL (integer and polynomial)	-
1	00	1100		0	SHA256H	FEAT_SHA256
		1010	0	0	VPMAX (integer)	-
1	01	0001		1	VBSL	-
		1010	0	1	VPMIN (integer)	-
		1010	1		UNALLOCATED	-
1	01	1100		0	SHA256H2	FEAT_SHA256
1	1x	1101		0	VABD (floating-point)	-
1	1x	1110		0	VCGT (register) — T2	-
1	1x	1110		1	VACGT	-
1	1x	1111	0	0	VPMIN (floating-point)	-
1	1x	1111		1	VMINNM	-
1		1000		0	VSUB (integer)	-
1	10	0001		1	VBIT	-
1		1000		1	VCEQ (register) — T1	-
1		1001		0	VMLS (integer)	-
1		1011		0	VQRDMULH	-
1	10	1100		0	SHA256SU1	FEAT_SHA256
1		1011		1	VQRDMLAH	FEAT_RDM
1	11	0001		1	VBIF	-
1		1100		1	VQRDMLSH	FEAT_RDM
1		1111	1	0	UNALLOCATED	-

Advanced SIMD two registers, or three registers of different lengths

These instructions are under Advanced SIMD data-processing.

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
111	op0	11111		op1		op2		op3		0	

op0	op1	op2	op3	Instruction details
0	11			VEXT (byte elements)
1	11	0x		Advanced SIMD two registers misc
1	11	10		VTBL, VTBX
1	11	11		Advanced SIMD duplicate (scalar)
	!= 11		0	Advanced SIMD three registers of different lengths
	!= 11		1	Advanced SIMD two registers and a scalar

Advanced SIMD two registers misc

These instructions are under Advanced SIMD two registers, or three registers of different lengths.

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	opc1	Vd	0	opc2	Q	M	0	Vm

size	opc1	opc2	Q	Instruction Details	Feature
	00	0000		VREV64	-
	00	0001		VREV32	-

Decode fields				Instruction Details	Feature
size	opc1	opc2	Q		
	00	0010		VREV16	-
	00	0011		UNALLOCATED	-
	00	010x		VPADDL	-
	00	0110	0	AESE	FEAT_AES
	00	0110	1	AESD	FEAT_AES
	00	0111	0	AESMC	FEAT_AES
	00	0111	1	AESIMC	FEAT_AES
	00	1000		VCLS	-
00	10	0000		VSWP	-
	00	1001		VCLZ	-
	00	1010		VCNT	-
	00	1011		VMVN (register)	-
00	10	1100	1	UNALLOCATED	-
	00	110x		VPADAL	-
	00	1110		VQABS	-
	00	1111		VQNEG	-
	01	x000		VCGT (immediate #0)	-
	01	x001		VCGE (immediate #0)	-
	01	x010		VCEQ (immediate #0)	-
	01	x011		VCLE (immediate #0)	-
	01	x100		VCLT (immediate #0)	-
	01	x110		VABS	-
	01	x111		VNEG	-
	01	0101	1	SHA1H	FEAT_SHA1
01	10	1100	1	VCVT (from single-precision to BFloat16, Advanced SIMD)	FEAT_AA32BF16
	10	0001		VTRN	-
	10	0010		VUZP	-
	10	0011		VZIP	-
	10	0100	0	VMOVN	-
	10	0100	1	VQMOVN, VQMOVUN — VQMOVUN	-
	10	0101		VQMOVN, VQMOVUN — VQMOVN	-
	10	0110	0	VSHLL	-
	10	0111	0	SHA1SU1	FEAT_SHA1
	10	0111	1	SHA256SU0	FEAT_SHA256
	10	1000		VRINTN (Advanced SIMD)	-
	10	1001		VRINTX (Advanced SIMD)	-
	10	1010		VRINTA (Advanced SIMD)	-
	10	1011		VRINTZ (Advanced SIMD)	-
10	10	1100	1	UNALLOCATED	-
	10	1100	0	VCVT (between half-precision and single-precision, Advanced SIMD) — single-precision to half-precision	-
	10	1101		VRINTM (Advanced SIMD)	-
	10	1110	0	VCVT (between half-precision and single-precision, Advanced SIMD) — half-precision to single-precision	-
	10	1110	1	UNALLOCATED	-
	10	1111		VRINTP (Advanced SIMD)	-
	11	000x		VCVTA (Advanced SIMD)	-
	11	001x		VCVTN (Advanced SIMD)	-

Decode fields size	opc1	opc2	Q	Instruction Details	Feature
	11	010x		VCVTP (Advanced SIMD)	-
	11	011x		VCVTM (Advanced SIMD)	-
	11	10x0		VRECPE	-
	11	10x1		VRSQRTE	-
11	10	1100	1	UNALLOCATED	-
	11	11xx		VCVT (between floating-point and integer, Advanced SIMD)	-

Advanced SIMD duplicate (scalar)

These instructions are under Advanced SIMD two registers, or three registers of different lengths.

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	imm4	Vd	1	1	opc	Q	M	0	Vm

Decode fields opc	Instruction Details
000	VDUP (scalar)
001	UNALLOCATED
01x	UNALLOCATED
1xx	UNALLOCATED

Advanced SIMD three registers of different lengths

These instructions are under Advanced SIMD two registers, or three registers of different lengths.

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	opc	N	0	M	0	Vm
										size								

The following constraints also apply to this encoding: size != 11 && size != 11

Decode fields U	opc	Instruction Details
	0000	VADDL
	0001	VADDW
	0010	VSUBL
0	0100	VADDHN
	0011	VSUBW
0	0110	VSUBHN
0	1001	VQDMLAL
	0101	VABAL
0	1011	VQDMLSL
0	1101	VQDMULL
	0111	VABDL (integer)
	1000	VMLAL (integer)
	1010	VMLSL (integer)
1	0100	VRADDHN
1	0110	VRSUBHN
	11x0	VMULL (integer and polynomial)
1	1001	UNALLOCATED
1	1011	UNALLOCATED

Decode fields U	opc	Instruction Details
1	1101	UNALLOCATED
	1111	UNALLOCATED

Advanced SIMD two registers and a scalar

These instructions are under Advanced SIMD two registers, or three registers of different lengths.

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	Q	1	1	1	1	1	D	!= 11 (size)	Vn	Vd	opc	N	1	M	0	Vm

The following constraints also apply to this encoding: size != 11 && size != 11

Decode fields Q	opc	Instruction Details	Feature
	000x	VMLA (by scalar)	-
0	0011	VQDMLAL	-
	0010	VMLAL (by scalar)	-
0	0111	VQDMLSL	-
	010x	VMLS (by scalar)	-
0	1011	VQDMULL	-
	0110	VMLSL (by scalar)	-
	100x	VMUL (by scalar)	-
1	0011	UNALLOCATED	-
	1010	VMULL (by scalar)	-
1	0111	UNALLOCATED	-
	1100	VQDMULH	-
	1101	VQRDMULH	-
1	1011	UNALLOCATED	-
	1110	VQRDMLAH	FEAT_RDM
	1111	VQRDMLSH	FEAT_RDM

Advanced SIMD shifts and immediate generation

These instructions are under Advanced SIMD data-processing.

15 14 13	12	11 10 9 8 7	6	5 ... 15 14 13 12 11 10 9 8	7 6 5	4	3 2 1 0
111		11111		op0		1	

Decode fields op0	Instruction details
000xxxxxxxxxxx0	Advanced SIMD one register and modified immediate
!= 000xxxxxxxxxxx0	Advanced SIMD two registers and shift amount

Advanced SIMD one register and modified immediate

These instructions are under Advanced SIMD shifts and immediate generation.

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	i	1	1	1	1	1	D	0	0	0	imm3	Vd	cmode	0	Q	op	1	imm4

Decode fields cmode	op	Instruction Details
0xx0	0	VMOV (immediate) — T1
0xx0	1	VMVN (immediate) — T1
0xx1	0	VORR (immediate) — T1
0xx1	1	VBIC (immediate) — T1
10x0	0	VMOV (immediate) — T3
10x0	1	VMVN (immediate) — T2
10x1	0	VORR (immediate) — T2
10x1	1	VBIC (immediate) — T2
11xx	0	VMOV (immediate) — T4
110x	1	VMVN (immediate) — T3
1110	1	VMOV (immediate) — T5
1111	1	UNALLOCATED

Advanced SIMD two registers and shift amount

These instructions are under Advanced SIMD shifts and immediate generation.

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	imm3H	imm3L	Vd	opc	L	Q	M	1	Vm

The following constraints also apply to this encoding: imm3H:imm3L:Vd:opc:L != 000xxxxxxxxxxx0

U	Decode fields imm3H:L	imm3L	opc	Q	Instruction Details
	!= 0000		0000		VSHR
	!= 0000		0001		VSRA
	!= 0000	000	1010	0	VMOVL
	!= 0000		0010		VRSHR
	!= 0000		0011		VRSRA
	!= 0000		0111		VQSHL, VQSHLU (immediate) — VQSHL
	!= 0000		1001	0	VQSHRN, VQSHRUN — VQSHRN
	!= 0000		1001	1	VQRSHRN, VQRSHRUN — VQRSHRN
	!= 0000		1010	0	VSHLL
	!= 0000		11xx		VCVT (between floating-point and fixed-point, Advanced SIMD)
0	!= 0000		0101		VSHL (immediate)
0	!= 0000		1000	0	VSHRN
0	!= 0000		1000	1	VRSHRN
1	!= 0000		0100		VSRI
1	!= 0000		0101		VSLI
1	!= 0000		0110		VQSHL, VQSHLU (immediate) — VQSHLU
1	!= 0000		1000	0	VQSHRN, VQSHRUN — VQSHRUN
1	!= 0000		1000	1	VQRSHRN, VQRSHRUN — VQRSHRUN

Advanced SIMD and System register load/store and 64-bit move

These instructions are under System register access, Advanced SIMD, and floating-point.

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
1110110	op0		1	op1	

Decode fields op0	op1	Instruction details
00x0	0x	Advanced SIMD and floating-point 64-bit move
00x0	11	System register 64-bit move
!= 00x0	0x	Advanced SIMD and floating-point load/store
!= 00x0	11	System register Load/Store
	10	UNALLOCATED

Advanced SIMD and floating-point 64-bit move

These instructions are under Advanced SIMD and System register load/store and 64-bit move.

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	D	0	op	Rt2	Rt	1	0	size	opc2	M	o3	Vm

Decode fields D	op	size	opc2	o3	Instruction Details
0					UNALLOCATED
1				0	UNALLOCATED
1		0x	00	1	UNALLOCATED
1			01		UNALLOCATED
1	0	10	00	1	VMOV (between two general-purpose registers and two single-precision registers) — from general-purpose registers
1	0	11	00	1	VMOV (between two general-purpose registers and a doubleword floating-point register) — from general-purpose registers
1			1x		UNALLOCATED
1	1	10	00	1	VMOV (between two general-purpose registers and two single-precision registers) — to general-purpose registers
1	1	11	00	1	VMOV (between two general-purpose registers and a doubleword floating-point register) — to general-purpose registers

System register 64-bit move

These instructions are under Advanced SIMD and System register load/store and 64-bit move.

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	D	0	L	Rt2	Rt	1	1	1	cp15	opc1	CRm

Decode fields D	L	Instruction Details
0		UNALLOCATED
1	0	MCRR
1	1	MRRC

Advanced SIMD and floating-point load/store

These instructions are under Advanced SIMD and System register load/store and 64-bit move.

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	P	U	D	W	L	Rn	Vd	1	0	size	imm8

The following constraints also apply to this encoding: P:U:D:W != 00x0

Decode fields							Instruction Details	Feature
P	U	W	L	Rn	size	imm8		
0	0	1					UNALLOCATED	-
0	1				0x		UNALLOCATED	-
0	1		0		10		VSTM, VSTMDB, VSTMIA — single-precision scalar	-
0	1		0		11	xxxxxxx0	VSTM, VSTMDB, VSTMIA — double-precision scalar	-
0	1		0		11	xxxxxxx1	FSTMDBX, FSTMIAX — Increment After	-
0	1		1		10		VLDM, VLDMDB, VLDMIA — single-precision scalar	-
0	1		1		11	xxxxxxx0	VLDM, VLDMDB, VLDMIA — double-precision scalar	-
0	1		1		11	xxxxxxx1	FLDM*X (FLDMDBX, FLDMIAX) — Increment After	-
1		0	0		01		VSTR — half-precision scalar	FEAT_FP16
1		0	0		10		VSTR — single-precision scalar	-
1		0	0		11		VSTR — double-precision scalar	-
1		0	1	!= 1111	01		VLDR (immediate) — half-precision scalar	FEAT_FP16
1		0	1	!= 1111	10		VLDR (immediate) — single-precision scalar	-
1		0	1	!= 1111	11		VLDR (immediate) — double-precision scalar	-
1	0	1			0x		UNALLOCATED	-
1	0	1	0		10		VSTM, VSTMDB, VSTMIA — single-precision scalar	-
1	0	1	0		11	xxxxxxx0	VSTM, VSTMDB, VSTMIA — double-precision scalar	-
1	0	1	0		11	xxxxxxx1	FSTMDBX, FSTMIAX — Decrement Before	-
1	0	1	1		10		VLDM, VLDMDB, VLDMIA — single-precision scalar	-
1	0	1	1		11	xxxxxxx0	VLDM, VLDMDB, VLDMIA — double-precision scalar	-
1	0	1	1		11	xxxxxxx1	FLDM*X (FLDMDBX, FLDMIAX) — Decrement Before	-
1		0	1	1111	01		VLDR (literal) — half-precision scalar	FEAT_FP16
1		0	1	1111	10		VLDR (literal) — single-precision scalar	-
1		0	1	1111	11		VLDR (literal) — double-precision scalar	-
1	1	1					UNALLOCATED	-

System register Load/Store

These instructions are under Advanced SIMD and System register load/store and 64-bit move.

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	P	U	D	W	L	Rn				CRd				1	1	1	cp15	imm8							

The following constraints also apply to this encoding: P:U:D:W != 00x0

Decode fields						Instruction Details
P:U:W	D	L	Rn	CRd	cp15	
!= 000				!= 0101	0	UNALLOCATED
!= 000	0	1	1111	0101	0	LDC (literal)

Decode fields P:U:W	D	L	Rn	CRd	cp15	Instruction Details
!= 000					1	UNALLOCATED
!= 000	1			0101	0	UNALLOCATED
0x1	0	0		0101	0	STC — post-indexed
0x1	0	1	!= 1111	0101	0	LDC (immediate) — post-indexed
010	0	0		0101	0	STC — unindexed
010	0	1	!= 1111	0101	0	LDC (immediate) — unindexed
1x0	0	0		0101	0	STC — offset
1x0	0	1	!= 1111	0101	0	LDC (immediate) — offset
1x1	0	0		0101	0	STC — pre-indexed
1x1	0	1	!= 1111	0101	0	LDC (immediate) — pre-indexed

Advanced SIMD and System register 32-bit move

These instructions are under System register access, Advanced SIMD, and floating-point.

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
11101110	op0		1	op1		1	

Decode fields op0	op1	Instruction details	Feature
000	000	UNALLOCATED	-
000	001	VMOV (between general-purpose register and half-precision)	FEAT_FP16
000	010	VMOV (between general-purpose register and single-precision)	-
001	010	UNALLOCATED	-
01x	010	UNALLOCATED	-
10x	010	UNALLOCATED	-
110	010	UNALLOCATED	-
111	010	Floating-point move special register	-
	011	Advanced SIMD 8/16/32-bit element move/duplicate	-
	10x	UNALLOCATED	-
	11x	System register 32-bit move	-

Floating-point move special register

These instructions are under Advanced SIMD and System register 32-bit move.

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	1	1	1	L	reg	Rt	1	0	1	0	(0)	(0)	(0)	1	(0)	(0)	(0)	(0)

Decode fields L	Instruction Details
0	VMSR
1	VMRS

Advanced SIMD 8/16/32-bit element move/duplicate

These instructions are under Advanced SIMD and System register 32-bit move.

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	opc1	L	Vn	Rt	1	0	1	1	N	opc2	1	(0)	(0)	(0)	(0)

Decode fields opc1	L	opc2	Instruction Details
0xx	0		VMOV (general-purpose register to scalar)
	1		VMOV (scalar to general-purpose register)
1xx	0	0x	VDUP (general-purpose register)
1xx	0	1x	UNALLOCATED

System register 32-bit move

These instructions are under Advanced SIMD and System register 32-bit move.

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	opc1	L	CRn	Rt	1	1	1	cp15	opc2	1	CRm

Decode fields L	Instruction Details
0	MCR
1	MRC

Floating-point data-processing

These instructions are under System register access, Advanced SIMD, and floating-point.

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
11101110	op0		10		op1		0	

Decode fields op0	op1	Instruction details
1x11	1	Floating-point data-processing (two registers)
1x11	0	Floating-point move immediate
!= 1x11		Floating-point data-processing (three registers)

Floating-point data-processing (two registers)

These instructions are under Floating-point data-processing.

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	1	D	1	1	o1	opc2	Vd	1	0	size	o3	1	M	0	Vm

Decode fields o1	opc2	size	o3	Instruction Details	Feature
		00		UNALLOCATED	-
0	000	01	0	UNALLOCATED	-
0	000	01	1	VABS — half-precision scalar	FEAT_FP16
0	000	10	0	VMOV (register) — single-precision scalar	-
0	000	10	1	VABS — single-precision scalar	-
0	000	11	0	VMOV (register) — double-precision scalar	-
0	000	11	1	VABS — double-precision scalar	-
0	001	01	0	VNEG — half-precision scalar	FEAT_FP16
0	001	01	1	VSQRT — half-precision scalar	FEAT_FP16
0	001	10	0	VNEG — single-precision scalar	-
0	001	10	1	VSQRT — single-precision scalar	-
0	001	11	0	VNEG — double-precision scalar	-
0	001	11	1	VSQRT — double-precision scalar	-

Decode fields o1	opc2	size	o3	Instruction Details	Feature
0	010	01		UNALLOCATED	-
0	010	10	0	VCVTB — half-precision to single-precision	-
0	010	10	1	VCVTT — half-precision to single-precision	-
0	010	11	0	VCVTB — half-precision to double-precision	-
0	010	11	1	VCVTT — half-precision to double-precision	-
0	011	01	0	VCVTB (BFloat16)	FEAT_AA32BF16
0	011	01	1	VCVTT (BFloat16)	FEAT_AA32BF16
0	011	10	0	VCVTB — single-precision to half-precision	-
0	011	10	1	VCVTT — single-precision to half-precision	-
0	011	11	0	VCVTB — double-precision to half-precision	-
0	011	11	1	VCVTT — double-precision to half-precision	-
0	100	01	0	VCMP	FEAT_FP16
0	100	01	1	VCMPE	FEAT_FP16
0	100	10	0	VCMP	-
0	100	10	1	VCMPE	-
0	100	11	0	VCMP	-
0	100	11	1	VCMPE	-
0	101	01	0	VCMP	FEAT_FP16
0	101	01	1	VCMPE	FEAT_FP16
0	101	10	0	VCMP	-
0	101	10	1	VCMPE	-
0	101	11	0	VCMP	-
0	101	11	1	VCMPE	-
0	110	01	0	VRINTR — half-precision scalar	FEAT_FP16
0	110	01	1	VRINTZ (floating-point) — half-precision scalar	FEAT_FP16
0	110	10	0	VRINTR — single-precision scalar	-
0	110	10	1	VRINTZ (floating-point) — single-precision scalar	-
0	110	11	0	VRINTR — double-precision scalar	-
0	110	11	1	VRINTZ (floating-point) — double-precision scalar	-
0	111	01	0	VRINTX (floating-point) — half-precision scalar	FEAT_FP16
0	111	01	1	UNALLOCATED	-
0	111	10	0	VRINTX (floating-point) — single-precision scalar	-
0	111	10	1	VCVT (between double-precision and single-precision) — single-precision to double-precision	-
0	111	11	0	VRINTX (floating-point) — double-precision scalar	-
0	111	11	1	VCVT (between double-precision and single-precision) — double-precision to single-precision	-
1	000	01		VCVT (integer to floating-point, floating-point) — half-precision scalar	FEAT_FP16
1	000	10		VCVT (integer to floating-point, floating-point) — single-precision scalar	-
1	000	11		VCVT (integer to floating-point, floating-point) — double-precision scalar	-
1	001	01		UNALLOCATED	-
1	001	10		UNALLOCATED	-
1	001	11	0	UNALLOCATED	-
1	001	11	1	VJCVT	FEAT_JSCVT
1	01x	01		VCVT (between floating-point and fixed-point, floating-point)	FEAT_FP16
1	01x	10		VCVT (between floating-point and fixed-point, floating-point)	-

Decode fields o1	opc2	size	o3	Instruction Details	Feature
1	01x	11		VCVT (between floating-point and fixed-point, floating-point)	-
1	100	01	0	VCVTR	FEAT_FP16
1	100	01	1	VCVT (floating-point to integer, floating-point)	FEAT_FP16
1	100	10	0	VCVTR	-
1	100	10	1	VCVT (floating-point to integer, floating-point)	-
1	100	11	0	VCVTR	-
1	100	11	1	VCVT (floating-point to integer, floating-point)	-
1	101	01	0	VCVTR	FEAT_FP16
1	101	01	1	VCVT (floating-point to integer, floating-point)	FEAT_FP16
1	101	10	0	VCVTR	-
1	101	10	1	VCVT (floating-point to integer, floating-point)	-
1	101	11	0	VCVTR	-
1	101	11	1	VCVT (floating-point to integer, floating-point)	-
1	11x	01		VCVT (between floating-point and fixed-point, floating-point)	FEAT_FP16
1	11x	10		VCVT (between floating-point and fixed-point, floating-point)	-
1	11x	11		VCVT (between floating-point and fixed-point, floating-point)	-

Floating-point move immediate

These instructions are under Floating-point data-processing.

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	1	D	1	1	imm4H	Vd	1	0	size	(0)	0	(0)	0	imm4L

Decode fields size	Instruction Details	Feature
00	UNALLOCATED	-
01	VMOV (immediate) — half-precision scalar	FEAT_FP16
10	VMOV (immediate) — single-precision scalar	-
11	VMOV (immediate) — double-precision scalar	-

Floating-point data-processing (three registers)

These instructions are under Floating-point data-processing.

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	o0	D	o1	Vn	Vd	1	0	size	N	o2	M	0	Vm

The following constraints also apply to this encoding: o0:D:o1 != 1x11

Decode fields o0:o1	size	o2	Instruction Details	Feature
!= 111	00		UNALLOCATED	-
000	01	0	VMLA (floating-point) — half-precision scalar	FEAT_FP16
000	01	1	VMLS (floating-point) — half-precision scalar	FEAT_FP16
000	10	0	VMLA (floating-point) — single-precision scalar	-
000	10	1	VMLS (floating-point) — single-precision scalar	-
000	11	0	VMLA (floating-point) — double-precision scalar	-
000	11	1	VMLS (floating-point) — double-precision scalar	-
001	01	0	VNMLS — half-precision scalar	FEAT_FP16

Decode fields o0:o1	size	o2	Instruction Details	Feature
001	01	1	VNMLA — half-precision scalar	FEAT_FP16
001	10	0	VNMLS — single-precision scalar	-
001	10	1	VNMLA — single-precision scalar	-
001	11	0	VNMLS — double-precision scalar	-
001	11	1	VNMLA — double-precision scalar	-
010	01	0	VMUL (floating-point) — half-precision scalar	FEAT_FP16
010	01	1	VNMUL — half-precision scalar	FEAT_FP16
010	10	0	VMUL (floating-point) — single-precision scalar	-
010	10	1	VNMUL — single-precision scalar	-
010	11	0	VMUL (floating-point) — double-precision scalar	-
010	11	1	VNMUL — double-precision scalar	-
011	01	0	VADD (floating-point) — half-precision scalar	FEAT_FP16
011	01	1	VSUB (floating-point) — half-precision scalar	FEAT_FP16
011	10	0	VADD (floating-point) — single-precision scalar	-
011	10	1	VSUB (floating-point) — single-precision scalar	-
011	11	0	VADD (floating-point) — double-precision scalar	-
011	11	1	VSUB (floating-point) — double-precision scalar	-
100	01	0	VDIV — half-precision scalar	FEAT_FP16
100	10	0	VDIV — single-precision scalar	-
100	11	0	VDIV — double-precision scalar	-
101	01	0	VFNMS — half-precision scalar	FEAT_FP16
101	01	1	VFNMA — half-precision scalar	FEAT_FP16
101	10	0	VFNMS — single-precision scalar	-
101	10	1	VFNMA — single-precision scalar	-
101	11	0	VFNMS — double-precision scalar	-
101	11	1	VFNMA — double-precision scalar	-
110	01	0	VFMA — half-precision scalar	FEAT_FP16
110	01	1	VFMS — half-precision scalar	FEAT_FP16
110	10	0	VFMA — single-precision scalar	-
110	10	1	VFMS — single-precision scalar	-
110	11	0	VFMA — double-precision scalar	-
110	11	1	VFMS — double-precision scalar	-

Additional Advanced SIMD and floating-point instructions

These instructions are under System register access, Advanced SIMD, and floating-point.

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
111111	op0		op1		1	op2	op3		op4		op5	

The following constraints also apply to this encoding: op0<2:1> != 11

Decode fields op0	op1	op2	op3	op4	op5	Instruction details
0xx			0x			Advanced SIMD three registers of the same length extension
100		0	!= 00	0	0	Floating-point conditional select

101	00xxxx	0	!= 00		0	Floating-point minNum/maxNum
101	110000	0	!= 00	1	0	Floating-point extraction and insertion
101	111xxx	0	!= 00	1	0	Floating-point directed convert to integer
10x		0	00			Advanced SIMD and floating-point multiply with accumulate
10x		1	0x			Advanced SIMD and floating-point dot product

Advanced SIMD three registers of the same length extension

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	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	0	op1		D	op2		Vn				Vd				1	op3	0	op4	N	Q	M	U	Vm			

Decode fields						Instruction Details	Feature
op1	op2	op3	op4	Q	U		
x1	0x	0	0	0	0	VCADD — 64-bit SIMD vector	FEAT_FCMA
x1	0x	0	0	0	1	UNALLOCATED	-
x1	0x	0	0	1	0	VCADD — 128-bit SIMD vector	FEAT_FCMA
x1	0x	0	0	1	1	UNALLOCATED	-
00	0x	0	0			UNALLOCATED	-
00	0x	0	1			UNALLOCATED	-
00	00	1	0	0	0	UNALLOCATED	-
00	00	1	0	0	1	UNALLOCATED	-
00	00	1	0	1	0	VMMLA	FEAT_AA32BF16
00	00	1	0	1	1	UNALLOCATED	-
00	00	1	1	0	0	VDOT (vector) — 64-bit SIMD vector	FEAT_AA32BF16
00	00	1	1	0	1	UNALLOCATED	-
00	00	1	1	1	0	VDOT (vector) — 128-bit SIMD vector	FEAT_AA32BF16
00	00	1	1	1	1	UNALLOCATED	-
00	01	1	0			UNALLOCATED	-
00	01	1	1			UNALLOCATED	-
00	10	0	0		1	VFMAL (vector)	FEAT_FHM
00	10	0	1			UNALLOCATED	-
00	10	1	0	0		UNALLOCATED	-
00	10	1	0	1	0	VSMMLA	FEAT_AA32I8MM
00	10	1	0	1	1	VUMMLA	FEAT_AA32I8MM
00	10	1	1	0	0	VSDOT (vector) — 64-bit SIMD vector	FEAT_DotProd
00	10	1	1	0	1	VUDOT (vector) — 64-bit SIMD vector	FEAT_DotProd
00	10	1	1	1	0	VSDOT (vector) — 128-bit SIMD vector	FEAT_DotProd
00	10	1	1	1	1	VUDOT (vector) — 128-bit SIMD vector	FEAT_DotProd
00	11	0	0		1	VFMAB, VFMAT (BFloat16, vector)	FEAT_AA32BF16
00	11	0	1			UNALLOCATED	-
00	11	1	0			UNALLOCATED	-
00	11	1	1			UNALLOCATED	-
01	10	0	0		1	VFMSL (vector)	FEAT_FHM
01	10	0	1			UNALLOCATED	-
01	10	1	0	0		UNALLOCATED	-

Decode fields						Instruction Details	Feature
op1	op2	op3	op4	Q	U		
01	10	1	0	1	0	VUSMMLA	FEAT_AA32I8MM
01	10	1	0	1	1	UNALLOCATED	-
01	10	1	1	0	0	VUSDOT (vector) — 64-bit SIMD vector	FEAT_AA32I8MM
01	10	1	1		1	UNALLOCATED	-
01	10	1	1	1	0	VUSDOT (vector) — 128-bit SIMD vector	FEAT_AA32I8MM
01	11	0	1			UNALLOCATED	-
01	11	1	0			UNALLOCATED	-
01	11	1	1			UNALLOCATED	-
	1x	0	0		0	VCMLA	FEAT_FCMA
10	11	0	1			UNALLOCATED	-
10	11	1	0			UNALLOCATED	-
10	11	1	1			UNALLOCATED	-
11	11	0	1			UNALLOCATED	-
11	11	1	0			UNALLOCATED	-
11	11	1	1			UNALLOCATED	-

Floating-point conditional select

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	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	cc	Vn	Vd	1	0	!= 00	N	0	M	0	Vm
															size					

The following constraints also apply to this encoding: size != 00 && size != 00

Decode fields	Instruction Details	Feature
size		
01	VSELEQ, VSELGE, VSELGT, VSELVS — half-precision scalar	FEAT_FP16
10	VSELEQ, VSELGE, VSELGT, VSELVS — single-precision scalar	-
11	VSELEQ, VSELGE, VSELGT, VSELVS — double-precision scalar	-

Floating-point minNum/maxNum

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	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	0	0	Vn	Vd	1	0	!= 00	N	op	M	0	Vm
																size					

The following constraints also apply to this encoding: size != 00 && size != 00

Decode fields		Instruction Details	Feature
size	op		
01	0	VMAXNM — half-precision scalar	FEAT_FP16
01	1	VMINNM — half-precision scalar	FEAT_FP16
10	0	VMAXNM — single-precision scalar	-
10	1	VMINNM — single-precision scalar	-
11	0	VMAXNM — double-precision scalar	-
11	1	VMINNM — double-precision scalar	-

Floating-point extraction and insertion

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	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	!= 00		op	1	M	0	Vm			
																						size									

The following constraints also apply to this encoding: size != 00 && size != 00

Decode fields size	op	Instruction Details	Feature
01		UNALLOCATED	-
10	0	VMOVX	FEAT_FP16
10	1	VINS	FEAT_FP16
11		UNALLOCATED	-

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	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	o1	RM		Vd				1	0	!= 00		op	1	M	0	Vm			
																						size									

The following constraints also apply to this encoding: size != 00 && size != 00

Decode fields o1	RM	size	op	Instruction Details	Feature
0		!= 00	1	UNALLOCATED	-
0	00	01	0	VRINTA (floating-point) — half-precision scalar	FEAT_FP16
0	00	10	0	VRINTA (floating-point) — single-precision scalar	-
0	00	11	0	VRINTA (floating-point) — double-precision scalar	-
0	01	01	0	VRINTN (floating-point) — half-precision scalar	FEAT_FP16
0	01	10	0	VRINTN (floating-point) — single-precision scalar	-
0	01	11	0	VRINTN (floating-point) — double-precision scalar	-
0	10	01	0	VRINTP (floating-point) — half-precision scalar	FEAT_FP16
0	10	10	0	VRINTP (floating-point) — single-precision scalar	-
0	10	11	0	VRINTP (floating-point) — double-precision scalar	-
0	11	01	0	VRINTM (floating-point) — half-precision scalar	FEAT_FP16
0	11	10	0	VRINTM (floating-point) — single-precision scalar	-
0	11	11	0	VRINTM (floating-point) — double-precision scalar	-
1	00	01		VCVTA (floating-point) — half-precision scalar	FEAT_FP16
1	00	10		VCVTA (floating-point) — single-precision scalar	-
1	00	11		VCVTA (floating-point) — double-precision scalar	-
1	01	01		VCVTN (floating-point) — half-precision scalar	FEAT_FP16
1	01	10		VCVTN (floating-point) — single-precision scalar	-
1	01	11		VCVTN (floating-point) — double-precision scalar	-
1	10	01		VCVTP (floating-point) — half-precision scalar	FEAT_FP16
1	10	10		VCVTP (floating-point) — single-precision scalar	-
1	10	11		VCVTP (floating-point) — double-precision scalar	-
1	11	01		VCVTM (floating-point) — half-precision scalar	FEAT_FP16
1	11	10		VCVTM (floating-point) — single-precision scalar	-

Decode fields o1	RM	size	op	Instruction Details	Feature
1	11	11		VCVTM (floating-point) — double-precision scalar	-

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	15 14 13 12	11	10	9	8	7	6	5	4	3 2 1 0
1	1	1	1	1	1	1	0	op1	D	op2	Vn	Vd	1	0	0	0	N	Q	M	U	Vm

Decode fields op1	op2	Q	U	Instruction Details	Feature
0			0	VCMLA (by element) — 128-bit SIMD vector of half-precision floating-point	FEAT_FCMA
0	00		1	VFMAL (by scalar)	FEAT_FHM
0	01		1	VFMSL (by scalar)	FEAT_FHM
0	10		1	UNALLOCATED	-
0	11		1	VFMAB, VFMAT (BFloat16, by scalar)	FEAT_AA32BF16
1		0	0	VCMLA (by element) — 64-bit SIMD vector of single-precision floating-point	FEAT_FCMA
1			1	UNALLOCATED	-
1		1	0	VCMLA (by element) — 128-bit SIMD vector of single-precision floating-point	FEAT_FCMA

Advanced SIMD and floating-point dot product

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	15 14 13 12	11	10	9	8	7	6	5	4	3 2 1 0
1	1	1	1	1	1	1	0	op1	D	op2	Vn	Vd	1	1	0	op4	N	Q	M	U	Vm

Decode fields op1	op2	op4	Q	U	Instruction Details	Feature
0	00	0			UNALLOCATED	-
0	00	1	0	0	VDOT (by element) — 64-bit SIMD vector	FEAT_AA32BF16
0	00	1		1	UNALLOCATED	-
0	00	1	1	0	VDOT (by element) — 128-bit SIMD vector	FEAT_AA32BF16
0	01	0			UNALLOCATED	-
0	10	0			UNALLOCATED	-
0	10	1	0	0	VSDOT (by element) — 64-bit SIMD vector	FEAT_DotProd
0	10	1	0	1	VUDOT (by element) — 64-bit SIMD vector	FEAT_DotProd
0	10	1	1	0	VSDOT (by element) — 128-bit SIMD vector	FEAT_DotProd
0	10	1	1	1	VUDOT (by element) — 128-bit SIMD vector	FEAT_DotProd
0	11				UNALLOCATED	-
1		0			UNALLOCATED	-
1	00	1	0	0	VUSDOT (by element) — 64-bit SIMD vector	FEAT_AA32I8MM
1	00	1	0	1	VSUDOT (by element) — 64-bit SIMD vector	FEAT_AA32I8MM
1	00	1	1	0	VUSDOT (by element) — 128-bit SIMD vector	FEAT_AA32I8MM
1	00	1	1	1	VSUDOT (by element) — 128-bit SIMD vector	FEAT_AA32I8MM
1	01	1			UNALLOCATED	-
1	1x	1			UNALLOCATED	-

Load/store multiple

These instructions are under 32-bit.

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	opc	0	W	L	Rn	P	M	register_list

Decode fields opc	L	Instruction Details
00	0	SRS, SRSDA, SRSDB, SRSIA, SRSIB — T1
00	1	RFE, RFEDA, RFEDB, RFEIA, RFEIB — T1
01	0	STM, STMIA, STMEA
01	1	LDM, LDMIA, LDMFD
10	0	STMDB, STMFD
10	1	LDMDB, LDMEA
11	0	SRS, SRSDA, SRSDB, SRSIA, SRSIB — T2
11	1	RFE, RFEDA, RFEDB, RFEIA, RFEIB — T2

Load/store dual, load/store exclusive, load-acquire/store-release, and table branch

These instructions are under 32-bit.

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
1110100	op0	op1	op2		op3	

The following constraints also apply to this encoding: op0<1> == 1

Decode fields op0	op1	op2	op3	Instruction details
0010				Load/store exclusive
0110	0		000	UNALLOCATED
0110	1		000	TBB, TBH
0110			01x	Load/store exclusive byte/half/dual
0110			1xx	Load-acquire / Store-release
0x11		!= 1111		Load/store dual (immediate, post-indexed)
1x10		!= 1111		Load/store dual (immediate)
1x11		!= 1111		Load/store dual (immediate, pre-indexed)
!= 0xx0		1111		LDRD (literal)

Load/store exclusive

These instructions are under Load/store dual, load/store exclusive, load-acquire/store-release, and table branch.

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	1	0	L	Rn	Rt	Rd	imm8

Decode fields L	Instruction Details
0	STREX
1	LDREX

Load/store exclusive byte/half/dual

These instructions are under Load/store dual, load/store exclusive, load-acquire/store-release, and table branch.

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	L	Rn				Rt				Rt2				0	1	sz		Rd			

Decode fields L	sz	Instruction Details
0	00	STREXB
0	01	STREXH
0	10	UNALLOCATED
0	11	STREXD
1	00	LDREXB
1	01	LDREXH
1	10	UNALLOCATED
1	11	LDREXD

Load-acquire / Store-release

These instructions are under Load/store dual, load/store exclusive, load-acquire/store-release, and table branch.

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	L	Rn				Rt				Rt2				1	op	sz		Rd			

Decode fields L	op	sz	Instruction Details
0	0	00	STLB
0	0	01	STLH
0	0	10	STL
0	0	11	UNALLOCATED
0	1	00	STLEXB
0	1	01	STLEXH
0	1	10	STLEX
0	1	11	STLEXD
1	0	00	LDAB
1	0	01	LDAH
1	0	10	LDA
1	0	11	UNALLOCATED
1	1	00	LDAEXB
1	1	01	LDAEXH
1	1	10	LDAEX
1	1	11	LDAEXD

Load/store dual (immediate, post-indexed)

These instructions are under Load/store dual, load/store exclusive, load-acquire/store-release, and table branch.

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	U	1	1	L	!= 1111				Rt				Rt2				imm8							
												Rn																			

The following constraints also apply to this encoding: Rn != 1111 && Rn != 1111

Decode fields L	Instruction Details
0	STRD (immediate)
1	LDRD (immediate)

Load/store dual (immediate)

These instructions are under Load/store dual, load/store exclusive, load-acquire/store-release, and table branch.

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	U	1	0	L	!= 1111				Rt				Rt2				imm8							
												Rn																			

The following constraints also apply to this encoding: Rn != 1111 && Rn != 1111

Decode fields L	Instruction Details
0	STRD (immediate)
1	LDRD (immediate)

Load/store dual (immediate, pre-indexed)

These instructions are under Load/store dual, load/store exclusive, load-acquire/store-release, and table branch.

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	U	1	1	L	!= 1111				Rt				Rt2				imm8							
												Rn																			

The following constraints also apply to this encoding: Rn != 1111 && Rn != 1111

Decode fields L	Instruction Details
0	STRD (immediate)
1	LDRD (immediate)

Data-processing (shifted register)

These instructions are under 32-bit.

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	op1				S	Rn				(0)	imm3			Rd				imm2		stype		Rm			

Decode fields op1	S	Rn	imm3:imm2:stype	Rd	Instruction Details
0000	0		!= 0000011		AND, ANDS (register) — AND, shift or rotate by value
0000	0		0000011		AND, ANDS (register) — AND, rotate right with extend
0000	1		!= 0000011	!= 1111	AND, ANDS (register) — ANDS, shift or rotate by value
0000	1		!= 0000011	1111	TST (register) — shift or rotate by value
0000	1		0000011	!= 1111	AND, ANDS (register) — ANDS, rotate right with extend
0000	1		0000011	1111	TST (register) — rotate right with extend
0001			!= 0000011		BIC, BICS (register) — shift or rotate by value
0001			0000011		BIC, BICS (register) — rotate right with extend
0010	0	!= 1111	!= 0000011		ORR, ORRS (register) — ORR, shift or rotate by value
0010	0	!= 1111	0000011		ORR, ORRS (register) — ORR, rotate right with extend
0010	0	1111	!= 0000011		MOV, MOVS (register) — MOV, shift or rotate by value

Decode fields					Instruction Details
op1	S	Rn	imm3:imm2:stype	Rd	
0010	0	1111	0000011		MOV, MOVS (register) — MOV, rotate right with extend
0010	1	!= 1111	!= 0000011		ORR, ORRS (register) — ORRS, shift or rotate by value
0010	1	!= 1111	0000011		ORR, ORRS (register) — ORRS, rotate right with extend
0010	1	1111	!= 0000011		MOV, MOVS (register) — MOVS, shift or rotate by value
0010	1	1111	0000011		MOV, MOVS (register) — MOVS, rotate right with extend
0011	0	!= 1111	!= 0000011		ORN, ORNS (register) — ORN, shift or rotate by value
0011	0	!= 1111	0000011		ORN, ORNS (register) — ORN, rotate right with extend
0011	0	1111	!= 0000011		MVN, MVNS (register) — MVN, shift or rotate by value
0011	0	1111	0000011		MVN, MVNS (register) — MVN, rotate right with extend
0011	1	!= 1111	!= 0000011		ORN, ORNS (register) — ORNS, shift or rotate by value
0011	1	!= 1111	0000011		ORN, ORNS (register) — ORNS, rotate right with extend
0011	1	1111	!= 0000011		MVN, MVNS (register) — MVNS, shift or rotate by value
0011	1	1111	0000011		MVN, MVNS (register) — MVNS, rotate right with extend
0100	0		!= 0000011		EOR, EORS (register) — EOR, shift or rotate by value
0100	0		0000011		EOR, EORS (register) — EOR, rotate right with extend
0100	1		!= 0000011	!= 1111	EOR, EORS (register) — EORS, shift or rotate by value
0100	1		!= 0000011	1111	TEQ (register) — shift or rotate by value
0100	1		0000011	!= 1111	EOR, EORS (register) — EORS, rotate right with extend
0100	1		0000011	1111	TEQ (register) — rotate right with extend
0101					UNALLOCATED
0110	0		xxxxx00		PKHBT, PKHTB — PKHBT
0110	0		xxxxx01		UNALLOCATED
0110	0		xxxxx10		PKHBT, PKHTB — PKHTB
0110	0		xxxxx11		UNALLOCATED
0111					UNALLOCATED
1000	0	!= 1101	!= 0000011		ADD, ADDS (register) — ADD, shift or rotate by value
1000	0	!= 1101	0000011		ADD, ADDS (register) — ADD, rotate right with extend
1000	0	1101	!= 0000011		ADD, ADDS (SP plus register) — ADD, shift or rotate by value
1000	0	1101	0000011		ADD, ADDS (SP plus register) — ADD, rotate right with extend
1000	1		!= 0000011	1111	CMN (register) — shift or rotate by value

Decode fields					Instruction Details
op1	S	Rn	imm3:imm2:stype	Rd	
1000	1	!= 1101	!= 0000011	!= 1111	ADD, ADDS (register) — ADDS, shift or rotate by value
1000	1	!= 1101	0000011	!= 1111	ADD, ADDS (register) — ADDS, rotate right with extend
1000	1		0000011	1111	CMN (register) — rotate right with extend
1000	1	1101	!= 0000011	!= 1111	ADD, ADDS (SP plus register) — ADDS, shift or rotate by value
1000	1	1101	0000011	!= 1111	ADD, ADDS (SP plus register) — ADDS, rotate right with extend
1001					UNALLOCATED
1010			!= 0000011		ADC, ADCS (register) — shift or rotate by value
1010			0000011		ADC, ADCS (register) — rotate right with extend
1011			!= 0000011		SBC, SBCS (register) — shift or rotate by value
1011			0000011		SBC, SBCS (register) — rotate right with extend
1100					UNALLOCATED
1101	0	!= 1101	!= 0000011		SUB, SUBS (register) — SUB, shift or rotate by value
1101	0	!= 1101	0000011		SUB, SUBS (register) — SUB, rotate right with extend
1101	0	1101	!= 0000011		SUB, SUBS (SP minus register) — SUB, shift or rotate by value
1101	0	1101	0000011		SUB, SUBS (SP minus register) — SUB, rotate right with extend
1101	1		!= 0000011	1111	CMP (register) — shift or rotate by value
1101	1	!= 1101	!= 0000011	!= 1111	SUB, SUBS (register) — SUBS, shift or rotate by value
1101	1	!= 1101	0000011	!= 1111	SUB, SUBS (register) — SUBS, rotate right with extend
1101	1		0000011	1111	CMP (register) — rotate right with extend
1101	1	1101	!= 0000011	!= 1111	SUB, SUBS (SP minus register) — SUBS, shift or rotate by value
1101	1	1101	0000011	!= 1111	SUB, SUBS (SP minus register) — SUBS, rotate right with extend
1110			!= 0000011		RSB, RSBS (register) — shift or rotate by value
1110			0000011		RSB, RSBS (register) — rotate right with extend
1111					UNALLOCATED

Branches and miscellaneous control

These instructions are under 32-bit.

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
11110	op0	op1	op2		1	op3		op4		op5	

Decode fields						Instruction details
op0	op1	op2	op3	op4	op5	
0	1110	0x	0x0		0	MSR (register)
0	1110	0x	0x0		1	MSR (Banked register)
0	1110	10	0x0	000		Hints
0	1110	10	0x0	!= 000		Change processor state

0	1110	11	0x0			Miscellaneous system
0	1111	00	0x0			BXJ
0	1111	01	0x0			Exception return
0	1111	1x	0x0		0	MRS
0	1111	1x	0x0		1	MRS (Banked register)
1	1110	00	000			DCPS
1	1110	00	010			UNALLOCATED
1	1110	01	0x0			UNALLOCATED
1	1110	1x	0x0			UNALLOCATED
1	1111	0x	0x0			UNALLOCATED
1	1111	1x	0x0			Exception generation
	!= 111x		0x0			B — T3
			0x1			B — T4
			1x0			BL, BLX (immediate) — T2
			1x1			BL, BLX (immediate) — T1

Hints

These instructions are under Branches and miscellaneous control.

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	0	(1)	(1)	(1)	(1)	1	0	(0)	0	(0)	0	0	0	hint				option			

Decode fields hint	option	Instruction Details	Feature
0000	0000	NOP	-
0000	0001	YIELD	-
0000	0010	WFE	-
0000	0011	WFI	-
0000	0100	SEV	-
0000	0101	SEVL	-
0000	011x	Reserved hint, behaves as NOP	-
0000	1xxx	Reserved hint, behaves as NOP	-
0001	0000	ESB	FEAT_RAS
0001	0001	Reserved hint, behaves as NOP	-
0001	0010	TSB CSYNC	FEAT_TRF
0001	0011	Reserved hint, behaves as NOP	-
0001	0100	CSDB	-
0001	0101	Reserved hint, behaves as NOP	-
0001	0110	CLRBHB	FEAT_CLRBHB
0001	0111	Reserved hint, behaves as NOP	-
0001	1xxx	Reserved hint, behaves as NOP	-
001x		Reserved hint, behaves as NOP	-
01xx		Reserved hint, behaves as NOP	-
10xx		Reserved hint, behaves as NOP	-
110x		Reserved hint, behaves as NOP	-
1110		Reserved hint, behaves as NOP	-
1111		DBG	-

Change processor state

These instructions are under [Branches and miscellaneous control](#).

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	0	(1)	(1)	(1)	(1)	1	0	(0)	0	(0)	imod		M	A	I	F	mode				

The following constraints also apply to this encoding: imod:M != 000

Decode fields imod	M	Instruction Details
00	1	CPS, CPSID, CPSIE — change mode
01		UNALLOCATED
10		CPS, CPSID, CPSIE — interrupt enable and change mode
11		CPS, CPSID, CPSIE — interrupt disable and change mode

Miscellaneous system

These instructions are under [Branches and miscellaneous control](#).

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	0	(0)	0	(1)	(1)	(1)	(1)	opc				option			

Decode fields opc	option	Instruction Details	Feature
000x		UNALLOCATED	-
0010		CLREX	-
0011		UNALLOCATED	-
0100	!= 0x00	DSB	-
0100	0000	SSBB	-
0100	0100	PSSBB	-
0101		DMB	-
0110		ISB	-
0111		SB	FEAT_SB
1xxx		UNALLOCATED	-

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	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	Rn				1	0	(0)	0	(1)	(1)	(1)	(1)	imm8							

Decode fields Rn:imm8	Instruction Details
!= 111000000000	SUB, SUBS (immediate)
111000000000	ERET

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	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	

Decode fields imm4	imm10	opt	Instruction Details
!= 1111			UNALLOCATED
1111	!= 0000000000		UNALLOCATED
1111	0000000000	00	UNALLOCATED
1111	0000000000	01	DCPS1
1111	0000000000	10	DCPS2
1111	0000000000	11	DCPS3

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	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

Decode fields o1	o2	Instruction Details
0	0	HVC
0	1	UNALLOCATED
1	0	SMC
1	1	UDF

Data-processing (modified immediate)

These instructions are under 32-bit.

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	op1	S	Rn	0	imm3	Rd	imm8

Decode fields op1	S	Rn	Rd	Instruction Details
0000	0			AND, ANDS (immediate) — AND
0000	1		!= 1111	AND, ANDS (immediate) — ANDS
0000	1		1111	TST (immediate)
0001				BIC, BICS (immediate)
0010	0	!= 1111		ORR, ORRS (immediate) — ORR
0010	0	1111		MOV, MOVS (immediate) — MOV
0010	1	!= 1111		ORR, ORRS (immediate) — ORRS
0010	1	1111		MOV, MOVS (immediate) — MOVS
0011	0	!= 1111		ORN, ORNS (immediate) — not flag setting
0011	0	1111		MVN, MVNS (immediate) — MVN
0011	1	!= 1111		ORN, ORNS (immediate) — flag setting
0011	1	1111		MVN, MVNS (immediate) — MVNS
0100	0			EOR, EORS (immediate) — EOR
0100	1		!= 1111	EOR, EORS (immediate) — EORS
0100	1		1111	TEQ (immediate)
0101				UNALLOCATED
011x				UNALLOCATED
1000	0	!= 1101		ADD, ADDS (immediate) — ADD
1000	0	1101		ADD, ADDS (SP plus immediate) — ADD
1000	1	!= 1101	!= 1111	ADD, ADDS (immediate) — ADDS

Decode fields				Instruction Details
op1	S	Rn	Rd	
1000	1	1101	!= 1111	ADD, ADDS (SP plus immediate) — ADDS
1000	1		1111	CMN (immediate)
1001				UNALLOCATED
1010				ADC, ADCS (immediate)
1011				SBC, SBCS (immediate)
1100				UNALLOCATED
1101	0	!= 1101		SUB, SUBS (immediate) — SUB
1101	0	1101		SUB, SUBS (SP minus immediate) — SUB
1101	1	!= 1101	!= 1111	SUB, SUBS (immediate) — SUBS
1101	1	1101	!= 1111	SUB, SUBS (SP minus immediate) — SUBS
1101	1		1111	CMP (immediate)
1110				RSB, RSBS (immediate)
1111				UNALLOCATED

Data-processing (plain binary immediate)

These instructions are under 32-bit.

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
11110		1	op0		op1	0		0	

Decode fields		Instruction details
op0	op1	
0	0x	Data-processing (simple immediate)
0	10	Move Wide (16-bit immediate)
0	11	UNALLOCATED
1		Saturate, Bitfield

Data-processing (simple immediate)

These instructions are under Data-processing (plain binary immediate).

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	1	0	o1	0	o2	0	Rn	0	imm3	Rd	imm8

Decode fields			Instruction Details
o1	o2	Rn	
0	0	!= 11x1	ADD, ADDS (immediate)
0	0	1101	ADD, ADDS (SP plus immediate)
0	0	1111	ADR — T3
0	1		UNALLOCATED
1	0		UNALLOCATED
1	1	!= 11x1	SUB, SUBS (immediate)
1	1	1101	SUB, SUBS (SP minus immediate)
1	1	1111	ADR — T2

Move Wide (16-bit immediate)

These instructions are under Data-processing (plain binary immediate).

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	1	0	o1	1	0	0	imm4				0	imm3			Rd				imm8							

Decode fields o1	Instruction Details
0	MOV, MOVS (immediate)
1	MOVT

Saturate, Bitfield

These instructions are under Data-processing (plain binary immediate).

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	op1			0	Rn				0	imm3			Rd				imm2		(0)	widthm1				

Decode fields op1	Rn	imm3:imm2	Instruction Details
000			SSAT — logical shift left
001		!= 00000	SSAT — arithmetic shift right
001		00000	SSAT16
010			SBFX
011	!= 1111		BFI
011	1111		BFC
100			USAT — logical shift left
101		!= 00000	USAT — arithmetic shift right
101		00000	USAT16
110			UBFX
111			UNALLOCATED

Advanced SIMD element or structure load/store

These instructions are under 32-bit.

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
11111001								op0			0									op1											

Decode fields op0	op1	Instruction details
0		Advanced SIMD load/store multiple structures
1	11	Advanced SIMD load single structure to all lanes
1	!= 11	Advanced SIMD load/store single structure to one lane

Advanced SIMD load/store multiple structures

These instructions are under Advanced SIMD element or structure load/store.

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	L	0	Rn				Vd				itype				size		align		Rm			

Decode fields L	itype	Rm	Instruction Details
0	000x	!= 11x1	VST4 (multiple 4-element structures)
0	000x	1101	VST4 (multiple 4-element structures)
0	000x	1111	VST4 (multiple 4-element structures)
0	0010	!= 11x1	VST1 (multiple single elements)

L	Decode fields itype	Rm	Instruction Details
0	0010	1101	VST1 (multiple single elements)
0	0010	1111	VST1 (multiple single elements)
0	0011	!= 11x1	VST2 (multiple 2-element structures)
0	0011	1101	VST2 (multiple 2-element structures)
0	0011	1111	VST2 (multiple 2-element structures)
0	010x	!= 11x1	VST3 (multiple 3-element structures)
0	010x	1101	VST3 (multiple 3-element structures)
0	010x	1111	VST3 (multiple 3-element structures)
0	0110	!= 11x1	VST1 (multiple single elements)
0	0110	1101	VST1 (multiple single elements)
0	0110	1111	VST1 (multiple single elements)
0	0111	!= 11x1	VST1 (multiple single elements)
0	0111	1101	VST1 (multiple single elements)
0	0111	1111	VST1 (multiple single elements)
0	100x	!= 11x1	VST2 (multiple 2-element structures)
0	100x	1101	VST2 (multiple 2-element structures)
0	100x	1111	VST2 (multiple 2-element structures)
0	1010	!= 11x1	VST1 (multiple single elements)
0	1010	1101	VST1 (multiple single elements)
0	1010	1111	VST1 (multiple single elements)
1	000x	!= 11x1	VLD4 (multiple 4-element structures)
1	000x	1101	VLD4 (multiple 4-element structures)
1	000x	1111	VLD4 (multiple 4-element structures)
1	0010	!= 11x1	VLD1 (multiple single elements)
1	0010	1101	VLD1 (multiple single elements)
1	0010	1111	VLD1 (multiple single elements)
1	0011	!= 11x1	VLD2 (multiple 2-element structures)
1	0011	1101	VLD2 (multiple 2-element structures)
1	0011	1111	VLD2 (multiple 2-element structures)
1	010x	!= 11x1	VLD3 (multiple 3-element structures)
1	010x	1101	VLD3 (multiple 3-element structures)
1	010x	1111	VLD3 (multiple 3-element structures)
	1011		UNALLOCATED
1	0110	!= 11x1	VLD1 (multiple single elements)
1	0110	1101	VLD1 (multiple single elements)
1	0110	1111	VLD1 (multiple single elements)
1	0111	!= 11x1	VLD1 (multiple single elements)
1	0111	1101	VLD1 (multiple single elements)
1	0111	1111	VLD1 (multiple single elements)
	11xx		UNALLOCATED
1	100x	!= 11x1	VLD2 (multiple 2-element structures)
1	100x	1101	VLD2 (multiple 2-element structures)
1	100x	1111	VLD2 (multiple 2-element structures)
1	1010	!= 11x1	VLD1 (multiple single elements)
1	1010	1101	VLD1 (multiple single elements)
1	1010	1111	VLD1 (multiple single elements)

Advanced SIMD load single structure to all lanes

These instructions are under Advanced SIMD element or structure load/store.

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	L	0	Rn				Vd				1	1	N		size		T	a	Rm			

Decode fields				Instruction Details
L	N	a	Rm	
0				UNALLOCATED
1	00		!= 11x1	VLD1 (single element to all lanes)
1	00		1101	VLD1 (single element to all lanes)
1	00		1111	VLD1 (single element to all lanes)
1	01		!= 11x1	VLD2 (single 2-element structure to all lanes)
1	01		1101	VLD2 (single 2-element structure to all lanes)
1	01		1111	VLD2 (single 2-element structure to all lanes)
1	10	0	!= 11x1	VLD3 (single 3-element structure to all lanes)
1	10	0	1101	VLD3 (single 3-element structure to all lanes)
1	10	0	1111	VLD3 (single 3-element structure to all lanes)
1	10	1		UNALLOCATED
1	11		!= 11x1	VLD4 (single 4-element structure to all lanes)
1	11		1101	VLD4 (single 4-element structure to all lanes)
1	11		1111	VLD4 (single 4-element structure to all lanes)

Advanced SIMD load/store single structure to one lane

These instructions are under Advanced SIMD element or structure load/store.

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	L	0	Rn				Vd				!= 11		N		index_align				Rm			
																				size											

The following constraints also apply to this encoding: size != 11 && size != 11

Decode fields				Instruction Details
L	size	N	Rm	
0	00	00	!= 11x1	VST1 (single element from one lane)
0	00	00	1101	VST1 (single element from one lane)
0	00	00	1111	VST1 (single element from one lane)
0	00	01	!= 11x1	VST2 (single 2-element structure from one lane)
0	00	01	1101	VST2 (single 2-element structure from one lane)
0	00	01	1111	VST2 (single 2-element structure from one lane)
0	00	10	!= 11x1	VST3 (single 3-element structure from one lane)
0	00	10	1101	VST3 (single 3-element structure from one lane)
0	00	10	1111	VST3 (single 3-element structure from one lane)
0	00	11	!= 11x1	VST4 (single 4-element structure from one lane)
0	00	11	1101	VST4 (single 4-element structure from one lane)
0	00	11	1111	VST4 (single 4-element structure from one lane)
0	01	00	!= 11x1	VST1 (single element from one lane)
0	01	00	1101	VST1 (single element from one lane)
0	01	00	1111	VST1 (single element from one lane)
0	01	01	!= 11x1	VST2 (single 2-element structure from one lane)
0	01	01	1101	VST2 (single 2-element structure from one lane)

Decode fields				Instruction Details
L	size	N	Rm	
0	01	01	1111	VST2 (single 2-element structure from one lane)
0	01	10	!= 11x1	VST3 (single 3-element structure from one lane)
0	01	10	1101	VST3 (single 3-element structure from one lane)
0	01	10	1111	VST3 (single 3-element structure from one lane)
0	01	11	!= 11x1	VST4 (single 4-element structure from one lane)
0	01	11	1101	VST4 (single 4-element structure from one lane)
0	01	11	1111	VST4 (single 4-element structure from one lane)
0	10	00	!= 11x1	VST1 (single element from one lane)
0	10	00	1101	VST1 (single element from one lane)
0	10	00	1111	VST1 (single element from one lane)
0	10	01	!= 11x1	VST2 (single 2-element structure from one lane)
0	10	01	1101	VST2 (single 2-element structure from one lane)
0	10	01	1111	VST2 (single 2-element structure from one lane)
0	10	10	!= 11x1	VST3 (single 3-element structure from one lane)
0	10	10	1101	VST3 (single 3-element structure from one lane)
0	10	10	1111	VST3 (single 3-element structure from one lane)
0	10	11	!= 11x1	VST4 (single 4-element structure from one lane)
0	10	11	1101	VST4 (single 4-element structure from one lane)
0	10	11	1111	VST4 (single 4-element structure from one lane)
1	00	00	!= 11x1	VLD1 (single element to one lane)
1	00	00	1101	VLD1 (single element to one lane)
1	00	00	1111	VLD1 (single element to one lane)
1	00	01	!= 11x1	VLD2 (single 2-element structure to one lane)
1	00	01	1101	VLD2 (single 2-element structure to one lane)
1	00	01	1111	VLD2 (single 2-element structure to one lane)
1	00	10	!= 11x1	VLD3 (single 3-element structure to one lane)
1	00	10	1101	VLD3 (single 3-element structure to one lane)
1	00	10	1111	VLD3 (single 3-element structure to one lane)
1	00	11	!= 11x1	VLD4 (single 4-element structure to one lane)
1	00	11	1101	VLD4 (single 4-element structure to one lane)
1	00	11	1111	VLD4 (single 4-element structure to one lane)
1	01	00	!= 11x1	VLD1 (single element to one lane)
1	01	00	1101	VLD1 (single element to one lane)
1	01	00	1111	VLD1 (single element to one lane)
1	01	01	!= 11x1	VLD2 (single 2-element structure to one lane)
1	01	01	1101	VLD2 (single 2-element structure to one lane)
1	01	01	1111	VLD2 (single 2-element structure to one lane)
1	01	10	!= 11x1	VLD3 (single 3-element structure to one lane)
1	01	10	1101	VLD3 (single 3-element structure to one lane)
1	01	10	1111	VLD3 (single 3-element structure to one lane)
1	01	11	!= 11x1	VLD4 (single 4-element structure to one lane)
1	01	11	1101	VLD4 (single 4-element structure to one lane)
1	01	11	1111	VLD4 (single 4-element structure to one lane)
1	10	00	!= 11x1	VLD1 (single element to one lane)
1	10	00	1101	VLD1 (single element to one lane)
1	10	00	1111	VLD1 (single element to one lane)
1	10	01	!= 11x1	VLD2 (single 2-element structure to one lane)

Decode fields L	size	N	Rm	Instruction Details
1	10	01	1101	VLD2 (single 2-element structure to one lane)
1	10	01	1111	VLD2 (single 2-element structure to one lane)
1	10	10	!= 11x1	VLD3 (single 3-element structure to one lane)
1	10	10	1101	VLD3 (single 3-element structure to one lane)
1	10	10	1111	VLD3 (single 3-element structure to one lane)
1	10	11	!= 11x1	VLD4 (single 4-element structure to one lane)
1	10	11	1101	VLD4 (single 4-element structure to one lane)
1	10	11	1111	VLD4 (single 4-element structure to one lane)

Load/store single

These instructions are under 32-bit.

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
1111100							op0				op1	op2								op3											

The following constraints also apply to this encoding: op0<1>:op1 != 10

Decode fields op0	op1	op2	op3	Instruction details
00		!= 1111	000000	Load/store, unsigned (register offset)
00		!= 1111	000001	UNALLOCATED
00		!= 1111	00001x	UNALLOCATED
00		!= 1111	0001xx	UNALLOCATED
00		!= 1111	001xxx	UNALLOCATED
00		!= 1111	01xxxx	UNALLOCATED
00		!= 1111	10x0xx	UNALLOCATED
00		!= 1111	10x1xx	Load/store, unsigned (immediate, post-indexed)
00		!= 1111	1100xx	Load/store, unsigned (negative immediate)
00		!= 1111	1110xx	Load/store, unsigned (unprivileged)
00		!= 1111	11x1xx	Load/store, unsigned (immediate, pre-indexed)
01		!= 1111		Load/store, unsigned (positive immediate)
0x		1111		Load, unsigned (literal)
10	1	!= 1111	000000	Load/store, signed (register offset)
10	1	!= 1111	000001	UNALLOCATED
10	1	!= 1111	00001x	UNALLOCATED
10	1	!= 1111	0001xx	UNALLOCATED
10	1	!= 1111	001xxx	UNALLOCATED
10	1	!= 1111	01xxxx	UNALLOCATED
10	1	!= 1111	10x0xx	UNALLOCATED
10	1	!= 1111	10x1xx	Load/store, signed (immediate, post-indexed)
10	1	!= 1111	1100xx	Load/store, signed (negative immediate)
10	1	!= 1111	1110xx	Load/store, signed (unprivileged)
10	1	!= 1111	11x1xx	Load/store, signed (immediate, pre-indexed)
11	1	!= 1111		Load/store, signed (positive immediate)
1x	1	1111		Load, signed (literal)

Load/store, unsigned (register offset)

These instructions are under Load/store single.

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	0	0	size	L	!= 1111	Rt	0	0	0	0	0	0	imm2	Rm
											Rn									

The following constraints also apply to this encoding: Rn != 1111 && Rn != 1111

Decode fields size	L	Rt	Instruction Details
00	0		STRB (register)
00	1	!= 1111	LDRB (register)
00	1	1111	PLD, PLDW (register) — preload read
01	0		STRH (register)
01	1	!= 1111	LDRH (register)
01	1	1111	PLD, PLDW (register) — preload write
10	0		STR (register)
10	1		LDR (register)
11			UNALLOCATED

Load/store, unsigned (immediate, post-indexed)

These instructions are under Load/store single.

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	0	0	size	L	!= 1111	Rt	1	0	U	1	imm8
											Rn						

The following constraints also apply to this encoding: Rn != 1111 && Rn != 1111

Decode fields size	L	Instruction Details
00	0	STRB (immediate)
00	1	LDRB (immediate)
01	0	STRH (immediate)
01	1	LDRH (immediate)
10	0	STR (immediate)
10	1	LDR (immediate)
11		UNALLOCATED

Load/store, unsigned (negative immediate)

These instructions are under Load/store single.

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	0	0	size	L	!= 1111	Rt	1	1	0	0	imm8
											Rn						

The following constraints also apply to this encoding: Rn != 1111 && Rn != 1111

Decode fields size	L	Rt	Instruction Details
00	0		STRB (immediate)

Decode fields			Instruction Details
size	L	Rt	
00	1	!= 1111	LDRB (immediate)
00	1	1111	PLD, PLDW (immediate) — preload read
01	0		STRH (immediate)
01	1	!= 1111	LDRH (immediate)
01	1	1111	PLD, PLDW (immediate) — preload write
10	0		STR (immediate)
10	1		LDR (immediate)
11			UNALLOCATED

Load/store, unsigned (unprivileged)

These instructions are under Load/store single.

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	0	0	size	L	!= 1111	Rt	1	1	1	0	imm8
											Rn						

The following constraints also apply to this encoding: Rn != 1111 && Rn != 1111

Decode fields		Instruction Details
size	L	
00	0	STRBT
00	1	LDRBT
01	0	STRHT
01	1	LDRHT
10	0	STRT
10	1	LDRT
11		UNALLOCATED

Load/store, unsigned (immediate, pre-indexed)

These instructions are under Load/store single.

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	0	0	size	L	!= 1111	Rt	1	1	U	1	imm8
											Rn						

The following constraints also apply to this encoding: Rn != 1111 && Rn != 1111

Decode fields		Instruction Details
size	L	
00	0	STRB (immediate)
00	1	LDRB (immediate)
01	0	STRH (immediate)
01	1	LDRH (immediate)
10	0	STR (immediate)
10	1	LDR (immediate)
11		UNALLOCATED

Load/store, unsigned (positive immediate)

These instructions are under Load/store single.

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	0	1	size		L	!= 1111				Rt				imm12											
												Rn																			

The following constraints also apply to this encoding: Rn != 1111 && Rn != 1111

Decode fields			Instruction Details
size	L	Rt	
00	0		STRB (immediate)
00	1	!= 1111	LDRB (immediate)
00	1	1111	PLD, PLDW (immediate) — preload read
01	0		STRH (immediate)
01	1	!= 1111	LDRH (immediate)
01	1	1111	PLD, PLDW (immediate) — preload write
10	0		STR (immediate)
10	1		LDR (immediate)

Load, unsigned (literal)

These instructions are under Load/store single.

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	0	U	size		L	1	1	1	1	Rt				imm12											

Decode fields			Instruction Details
size	L	Rt	
0x	1	1111	PLD (literal)
00	1	!= 1111	LDRB (literal)
01	1	!= 1111	LDRH (literal)
10	1		LDR (literal)
11			UNALLOCATED

Load/store, signed (register offset)

These instructions are under Load/store single.

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	size		1	!= 1111				Rt				0	0	0	0	0	0	imm2		Rm			
												Rn																			

The following constraints also apply to this encoding: Rn != 1111 && Rn != 1111

Decode fields		Instruction Details
size	Rt	
00	!= 1111	LDRSB (register)
00	1111	PLI (register)
01	!= 1111	LDRSH (register)
01	1111	Reserved hint, behaves as NOP
1x		UNALLOCATED

Load/store, signed (immediate, post-indexed)

These instructions are under Load/store single.

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	size		1	!= 1111				Rt				1	0	U	1	imm8							
												Rn																			

The following constraints also apply to this encoding: Rn != 1111 && Rn != 1111

Decode fields size	Instruction Details
00	LDRSB (immediate)
01	LDRSH (immediate)
1x	UNALLOCATED

Load/store, signed (negative immediate)

These instructions are under Load/store single.

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	size		1	!= 1111				Rt				1	1	0	0	imm8							
												Rn																			

The following constraints also apply to this encoding: Rn != 1111 && Rn != 1111

Decode fields size	 Rt	Instruction Details
00	!= 1111	LDRSB (immediate)
00	1111	PLI (immediate, literal)
01	!= 1111	LDRSH (immediate)
01	1111	Reserved hint, behaves as NOP
1x		UNALLOCATED

Load/store, signed (unprivileged)

These instructions are under Load/store single.

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	size		1	!= 1111				Rt				1	1	1	0	imm8							
												Rn																			

The following constraints also apply to this encoding: Rn != 1111 && Rn != 1111

Decode fields size	Instruction Details
00	LDRSBT
01	LDRSHT
1x	UNALLOCATED

Load/store, signed (immediate, pre-indexed)

These instructions are under Load/store single.

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	size		1	!= 1111				Rt				1	1	U	1	imm8							
												Rn																			

The following constraints also apply to this encoding: Rn != 1111 && Rn != 1111

Decode fields size	Instruction Details
00	LDRSB (immediate)
01	LDRSH (immediate)
1x	UNALLOCATED

Load/store, signed (positive immediate)

These instructions are under Load/store single.

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	size	1	!= 1111 Rn	Rt	imm12

The following constraints also apply to this encoding: Rn != 1111 && Rn != 1111

Decode fields size	 Rt	Instruction Details
00	!= 1111	LDRSB (immediate)
00	1111	PLI (immediate, literal)
01	!= 1111	LDRSH (immediate)
01	1111	Reserved hint, behaves as NOP

Load, signed (literal)

These instructions are under Load/store single.

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	U	size	1	1	1	1	1	Rt	imm12

Decode fields size	 Rt	Instruction Details
00	!= 1111	LDRSB (literal)
00	1111	PLI (immediate, literal)
01	!= 1111	LDRSH (literal)
01	1111	Reserved hint, behaves as NOP
1x		UNALLOCATED

Data-processing (register)

These instructions are under 32-bit.

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
11111010	op0		op1		op2	

Decode fields op0	 op1	 op2	Instruction details
0	1111	0000	MOV, MOVS (register-shifted register) — T2, Flag setting
0	1111	0001	UNALLOCATED
0	1111	001x	UNALLOCATED
0	1111	01xx	UNALLOCATED
0	1111	1xxx	Register extends
1	1111	0xxx	Parallel add-subtract

1	1111	10xx	Data-processing (two source registers)
1	1111	11xx	UNALLOCATED
	!= 1111		UNALLOCATED

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	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	op1		U	Rn				1	1	1	1	Rd				1	(0)	rotate		Rm			

Decode fields op1	U	Rn	Instruction Details
00	0	!= 1111	SXTAH
00	0	1111	SXTH
00	1	!= 1111	UXTAH
00	1	1111	UXTH
01	0	!= 1111	SXTAB16
01	0	1111	SXTB16
01	1	!= 1111	UXTAB16
01	1	1111	UXTB16
10	0	!= 1111	SXTAB
10	0	1111	SXTB
10	1	!= 1111	UXTAB
10	1	1111	UXTB
11			UNALLOCATED

Parallel add-subtract

These instructions are under Data-processing (register).

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	0	1	op1			Rn				1	1	1	1	Rd				0	U	H	S	Rm			

Decode fields op1	U	H	S	Instruction Details
000	0	0	0	SADD8
000	0	0	1	QADD8
000	0	1	0	SHADD8
000	0	1	1	UNALLOCATED
000	1	0	0	UADD8
000	1	0	1	UQADD8
000	1	1	0	UHADD8
000	1	1	1	UNALLOCATED
001	0	0	0	SADD16
001	0	0	1	QADD16
001	0	1	0	SHADD16
001	0	1	1	UNALLOCATED
001	1	0	0	UADD16
001	1	0	1	UQADD16
001	1	1	0	UHADD16
001	1	1	1	UNALLOCATED

Decode fields op1	U	H	S	Instruction Details
010	0	0	0	SASX
010	0	0	1	QASX
010	0	1	0	SHASX
010	0	1	1	UNALLOCATED
010	1	0	0	UASX
010	1	0	1	UQASX
010	1	1	0	UHASX
010	1	1	1	UNALLOCATED
100	0	0	0	SSUB8
100	0	0	1	QSUB8
100	0	1	0	SHSUB8
100	0	1	1	UNALLOCATED
100	1	0	0	USUB8
100	1	0	1	UQSUB8
100	1	1	0	UHSUB8
100	1	1	1	UNALLOCATED
101	0	0	0	SSUB16
101	0	0	1	QSUB16
101	0	1	0	SHSUB16
101	0	1	1	UNALLOCATED
101	1	0	0	USUB16
101	1	0	1	UQSUB16
101	1	1	0	UHSUB16
101	1	1	1	UNALLOCATED
110	0	0	0	SSAX
110	0	0	1	QSAX
110	0	1	0	SHSAX
110	0	1	1	UNALLOCATED
110	1	0	0	USAX
110	1	0	1	UQSAX
110	1	1	0	UHSAX
110	1	1	1	UNALLOCATED
111				UNALLOCATED

Data-processing (two source registers)

These instructions are under Data-processing (register).

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	0	1	op1	Rn	1	1	1	1	Rd	1	0	op2	Rm

Decode fields op1	op2	Instruction Details	Feature
000	00	QADD	-
000	01	QDADD	-
000	10	QSUB	-
000	11	QDSUB	-
001	00	REV	-
001	01	REV16	-

Decode fields op1	op2	Instruction Details	Feature
001	10	RBIT	-
001	11	REVSH	-
010	00	SEL	-
010	01	UNALLOCATED	-
010	1x	UNALLOCATED	-
011	00	CLZ	-
011	01	UNALLOCATED	-
011	1x	UNALLOCATED	-
100	00	CRC32 — CRC32B	FEAT_CRC32
100	01	CRC32 — CRC32H	FEAT_CRC32
100	10	CRC32 — CRC32W	FEAT_CRC32
100	11	CONSTRAINED UNPREDICTABLE	-
101	00	CRC32C — CRC32CB	FEAT_CRC32
101	01	CRC32C — CRC32CH	FEAT_CRC32
101	10	CRC32C — CRC32CW	FEAT_CRC32
101	11	CONSTRAINED UNPREDICTABLE	-
11x		UNALLOCATED	-

The behavior of the CONSTRAINED UNPREDICTABLE encodings in this table is described in *CONSTRAINED UNPREDICTABLE behavior for A32 and T32 instruction encodings*

Multiply, multiply accumulate, and absolute difference

These instructions are under 32-bit.

15–7	6–15	7–6	5–0
111110110		op0	

Decode fields op0	Instruction details
00	Multiply and absolute difference
01	UNALLOCATED
1x	UNALLOCATED

Multiply and absolute difference

These instructions are under Multiply, multiply accumulate, and absolute difference.

15	14	13	12	11	10	9	8	7	6–4	3–0	15–12	11–8	7	6	5–4	3–0
1	1	1	1	1	0	1	1	0	op1	Rn	Ra	Rd	0	0	op2	Rm

Decode fields op1	Ra	op2	Instruction Details
000	!= 1111	00	MLA, MLAS
000		01	MLS
000		1x	UNALLOCATED
000	1111	00	MUL, MULS
001	!= 1111	00	SMLABB, SMLABT, SMLATB, SMLATT — SMLABB
001	!= 1111	01	SMLABB, SMLABT, SMLATB, SMLATT — SMLABT
001	!= 1111	10	SMLABB, SMLABT, SMLATB, SMLATT — SMLATB

Decode fields op1	Ra	op2	Instruction Details
001	!= 1111	11	SMLABB, SMLABT, SMLATB, SMLATT — SMLATT
001	1111	00	SMULBB, SMULBT, SMULTB, SMULTT — SMULBB
001	1111	01	SMULBB, SMULBT, SMULTB, SMULTT — SMULBT
001	1111	10	SMULBB, SMULBT, SMULTB, SMULTT — SMULTB
001	1111	11	SMULBB, SMULBT, SMULTB, SMULTT — SMULTT
010	!= 1111	00	SMLAD, SMLADX — SMLAD
010	!= 1111	01	SMLAD, SMLADX — SMLADX
010		1x	UNALLOCATED
010	1111	00	SMUAD, SMUADX — SMUAD
010	1111	01	SMUAD, SMUADX — SMUADX
011	!= 1111	00	SMLAWB, SMLAWT — SMLAWB
011	!= 1111	01	SMLAWB, SMLAWT — SMLAWT
011		1x	UNALLOCATED
011	1111	00	SMULWB, SMULWT — SMULWB
011	1111	01	SMULWB, SMULWT — SMULWT
100	!= 1111	00	SMLSD, SMLSDX — SMLSD
100	!= 1111	01	SMLSD, SMLSDX — SMLSDX
100		1x	UNALLOCATED
100	1111	00	SMUSD, SMUSDX — SMUSD
100	1111	01	SMUSD, SMUSDX — SMUSDX
101	!= 1111	00	SMMLA, SMMLAR — SMMLA
101	!= 1111	01	SMMLA, SMMLAR — SMMLAR
101		1x	UNALLOCATED
101	1111	00	SMMUL, SMMULR — SMMUL
101	1111	01	SMMUL, SMMULR — SMMULR
110		00	SMMLS, SMMLSR — SMMLS
110		01	SMMLS, SMMLSR — SMMLSR
110		1x	UNALLOCATED
111	!= 1111	00	USADA8
111		01	UNALLOCATED
111		1x	UNALLOCATED
111	1111	00	USAD8

Long multiply and divide

These instructions are under 32-bit.

15	14	13	12	11	10	9	8	7	6-4	3-0	15-12	11-8	7-4	3-0
1	1	1	1	1	0	1	1	1	op1	Rn	RdLo	RdHi	op2	Rm

Decode fields op1	op2	Instruction Details
000	!= 0000	UNALLOCATED
000	0000	SMULL, SMULLS
001	!= 1111	UNALLOCATED
001	1111	SDIV
010	!= 0000	UNALLOCATED
010	0000	UMULL, UMULLS
011	!= 1111	UNALLOCATED

Decode fields op1	op2	Instruction Details
011	1111	UDIV
100	0000	SMLAL, SMLALS
100	0001	UNALLOCATED
100	001x	UNALLOCATED
100	01xx	UNALLOCATED
100	1000	SMLALBB, SMLALBT, SMLALTB, SMLALTT — SMLALBB
100	1001	SMLALBB, SMLALBT, SMLALTB, SMLALTT — SMLALBT
100	1010	SMLALBB, SMLALBT, SMLALTB, SMLALTT — SMLALTB
100	1011	SMLALBB, SMLALBT, SMLALTB, SMLALTT — SMLALTT
100	1100	SMLALD, SMLALDX — SMLALD
100	1101	SMLALD, SMLALDX — SMLALDX
100	111x	UNALLOCATED
101	0xxx	UNALLOCATED
101	10xx	UNALLOCATED
101	1100	SMLSLD, SMLSLDX — SMLSLD
101	1101	SMLSLD, SMLSLDX — SMLSLDX
101	111x	UNALLOCATED
110	0000	UMLAL, UMLALS
110	0001	UNALLOCATED
110	001x	UNALLOCATED
110	010x	UNALLOCATED
110	0110	UMAAL
110	0111	UNALLOCATED
110	1xxx	UNALLOCATED
111		UNALLOCATED

Shared Pseudocode Functions

This page displays common pseudocode functions shared by many pages

Pseudocodes

Library pseudocode for aarch32/at/AArch32.AT

```
// AArch32.AT()
// ============
// Perform address translation as per AT instructions.

AArch32.AT(bits(32) vaddress, TranslationStage stage_in, bits(2) el, ATAccess ataccess)
    TranslationStage stage = stage_in;
    SecurityState ss;
    Regime regime;
    boolean eae;

    // ATS1Hx instructions
    if el == EL2 then
        regime = Regime_EL2;
        eae = TRUE;
        ss = SS_NonSecure;

    // ATS1Cxx instructions
    elsif stage == TranslationStage_1 || (stage == TranslationStage_12 && !HaveEL(EL2)) then
        stage = TranslationStage_1;
        ss = SecurityStateAtEL(PSTATE.EL);
        regime = if ss == SS_Secure && ELUsingAArch32(EL3) then Regime_EL30 else Regime_EL10;
        eae = TTBCR.EAE == '1';

    // ATS12NSOxx instructions
    else
        regime = Regime_EL10;
        eae = if HaveAArch32EL(EL3) then TTBCR_NS.EAE == '1' else TTBCR.EAE == '1';
        ss = SS_NonSecure;

    AddressDescriptor addrdesc;
    SDFType sdftype;
    boolean aligned = TRUE;
    bit supersection = '0';

    boolean write = ataccess IN {ATAccess_WritePAN, ATAccess_Write};
    boolean pan   = ataccess IN {ATAccess_WritePAN, ATAccess_ReadPAN};
    accdesc = CreateAccDescAT(ss, el, write, pan);

    // Prepare fault fields in case a fault is detected
    fault = NoFault(accdesc);

    if eae then
        (fault, addrdesc) = AArch32.S1TranslateLD(fault, regime, vaddress, aligned, accdesc);
    else
        (fault, addrdesc, sdftype) = AArch32.S1TranslateSD(fault, regime, vaddress, aligned,
                                                            accdesc);
        supersection = if sdftype == SDFType_Supersection then '1' else '0';

    // ATS12NSOxx instructions
    if stage == TranslationStage_12 && fault.statuscode == Fault_None then
        (fault, addrdesc) = AArch32.S2Translate(fault, addrdesc, aligned, accdesc);

    if fault.statuscode != Fault_None then
        // Take exception on External abort or when a fault occurs on translation table walk
        if IsExternalAbort(fault) || (PSTATE.EL == EL1 && EL2Enabled() && fault.s2fs1walk) then
            PAR = bits(64) UNKNOWN;
            AArch32.Abort(vaddress, fault);

    addrdesc.fault = fault;

    if (eae || (stage == TranslationStage_12 && (HCR.VM == '1' || HCR.DC == '1'))
            || (stage == TranslationStage_1 && el != EL2 && PSTATE.EL == EL2)) then
        AArch32.EncodePARLD(addrdesc, ss);
    else
        AArch32.EncodePARSD(addrdesc, supersection, ss);
    return;
```

Library pseudocode for aarch32/at/AArch32.EncodePARLD

```
// AArch32.EncodePARLD()
// =====================
// Returns 64-bit format PAR on address translation instruction.

AArch32.EncodePARLD(AddressDescriptor addrdesc, SecurityState ss)

    if !IsFault(addrdesc) then
        bit ns;
        if ss == SS_NonSecure then
            ns = bit UNKNOWN;
        elsif addrdesc.paddress.paspace == PAS_Secure then
            ns = '0';
        else
            ns = '1';
        PAR.F      = '0';
        PAR.SH     = ReportedPARShareability(PAREncodeShareability(addrdesc.memattrs));
        PAR.NS     = ns;
        PAR<10>    = bit IMPLEMENTATION_DEFINED "Non-Faulting PAR";               // IMPDEF
        PAR.LPAE   = '1';
        PAR.PA     = addrdesc.paddress.address<39:12>;
        PAR.ATTR   = ReportedPARAttrs(EncodePARAttrs(addrdesc.memattrs));
    else
        PAR.F      = '1';
        PAR.FST    = AArch32.PARFaultStatusLD(addrdesc.fault);
        PAR.S2WLK  = if addrdesc.fault.s2fs1walk then '1' else '0';
        PAR.FSTAGE = if addrdesc.fault.secondstage then '1' else '0';
        PAR.LPAE   = '1';
        PAR<63:48> = bits(16) IMPLEMENTATION_DEFINED "Faulting PAR";              // IMPDEF
    return;
```

Library pseudocode for aarch32/at/AArch32.EncodePARSD

```
// AArch32.EncodePARSD()
// =====================
// Returns 32-bit format PAR on address translation instruction.

AArch32.EncodePARSD(AddressDescriptor addrdesc_in, bit supersection, SecurityState ss)
    AddressDescriptor addrdesc = addrdesc_in;
    if !IsFault(addrdesc) then
        if (addrdesc.memattrs.memtype == MemType_Device ||
                (addrdesc.memattrs.inner.attrs == MemAttr_NC &&
                 addrdesc.memattrs.outer.attrs == MemAttr_NC)) then
            addrdesc.memattrs.shareability = Shareability_OSH;
        bit ns;
        if ss == SS_NonSecure then
            ns = bit UNKNOWN;
        elsif addrdesc.paddress.paspace == PAS_Secure then
            ns = '0';
        else
            ns = '1';
        bits(2) sh = if addrdesc.memattrs.shareability != Shareability_NSH then '01' else '00';
        PAR.F      = '0';
        PAR.SS     = supersection;
        PAR.Outer  = AArch32.ReportedOuterAttrs(AArch32.PAROuterAttrs(addrdesc.memattrs));
        PAR.Inner  = AArch32.ReportedInnerAttrs(AArch32.PARInnerAttrs(addrdesc.memattrs));
        PAR.SH     = ReportedPARShareability(sh);
        PAR<8>     = bit IMPLEMENTATION_DEFINED "Non-Faulting PAR";               // IMPDEF
        PAR.NS     = ns;
        PAR.NOS    = if addrdesc.memattrs.shareability == Shareability_OSH then '0' else '1';
        PAR.LPAE   = '0';
        PAR.PA     = addrdesc.paddress.address<39:12>;
    else
        PAR.F      = '1';
        PAR.FST    = AArch32.PARFaultStatusSD(addrdesc.fault);
        PAR.LPAE   = '0';
        PAR<31:16> = bits(16) IMPLEMENTATION_DEFINED "Faulting PAR";              // IMPDEF
    return;
```

Library pseudocode for aarch32/at/AArch32.PARFaultStatusLD

```
// AArch32.PARFaultStatusLD()
// ==========================
// Fault status field decoding of 64-bit PAR

bits(6) AArch32.PARFaultStatusLD(FaultRecord fault)
    bits(6) syndrome;

    if fault.statuscode == Fault_Domain then
        // Report Domain fault
        assert fault.level IN {1,2};
        syndrome<1:0> = if fault.level == 1 then '01' else '10';
        syndrome<5:2> = '1111';
    else
        syndrome = EncodeLDFSC(fault.statuscode, fault.level);
    return syndrome;
```

Library pseudocode for aarch32/at/AArch32.PARFaultStatusSD

```
// AArch32.PARFaultStatusSD()
// ==========================
// Fault status field decoding of 32-bit PAR.

bits(6) AArch32.PARFaultStatusSD(FaultRecord fault)
    bits(6) syndrome;

    syndrome<5> = if IsExternalAbort(fault) then fault.extflag else '0';
    syndrome<4:0> = EncodeSDFSC(fault.statuscode, fault.level);
    return syndrome;
```

Library pseudocode for aarch32/at/AArch32.PARInnerAttrs

```
// AArch32.PARInnerAttrs()
// =======================
// Convert orthogonal attributes and hints to 32-bit PAR Inner field.

bits(3) AArch32.PARInnerAttrs(MemoryAttributes memattrs)
    bits(3) result;

    if memattrs.memtype == MemType_Device then
        if memattrs.device == DeviceType_nGnRnE then
            result = '001'; // Non-cacheable
        elsif memattrs.device == DeviceType_nGnRE then
            result = '011'; // Non-cacheable
    else
        MemAttrHints inner = memattrs.inner;
        if inner.attrs == MemAttr_NC then
            result = '000'; // Non-cacheable
        elsif inner.attrs == MemAttr_WB && inner.hints<0> == '1' then
            result = '101'; // Write-Back, Write-Allocate
        elsif inner.attrs == MemAttr_WT then
            result = '110'; // Write-Through
        elsif inner.attrs == MemAttr_WB && inner.hints<0> == '0' then
            result = '111'; // Write-Back, no Write-Allocate
    return result;
```

Library pseudocode for aarch32/at/AArch32.PAROuterAttrs

```
// AArch32.PAROuterAttrs()
// =======================
// Convert orthogonal attributes and hints to 32-bit PAR Outer field.

bits(2) AArch32.PAROuterAttrs(MemoryAttributes memattrs)
    bits(2) result;

    if memattrs.memtype == MemType_Device then
        result = bits(2) UNKNOWN;
    else
        MemAttrHints outer = memattrs.outer;
        if outer.attrs == MemAttr_NC then
            result = '00'; // Non-cacheable
        elsif outer.attrs == MemAttr_WB && outer.hints<0> == '1' then
            result = '01'; // Write-Back, Write-Allocate
        elsif outer.attrs == MemAttr_WT && outer.hints<0> == '0' then
            result = '10'; // Write-Through, no Write-Allocate
        elsif outer.attrs == MemAttr_WB && outer.hints<0> == '0' then
            result = '11'; // Write-Back, no Write-Allocate
    return result;
```

Library pseudocode for aarch32/at/AArch32.ReportedInnerAttrs

```
// AArch32.ReportedInnerAttrs()
// ============================
// The value returned in this field can be the resulting attribute, as determined by any permitted
// implementation choices and any applicable configuration bits, instead of the value that appears
// in the translation table descriptor.

bits(3) AArch32.ReportedInnerAttrs(bits(3) attrs);
```

Library pseudocode for aarch32/at/AArch32.ReportedOuterAttrs

```
// AArch32.ReportedOuterAttrs()
// ============================
// The value returned in this field can be the resulting attribute, as determined by any permitted
// implementation choices and any applicable configuration bits, instead of the value that appears
// in the translation table descriptor.

bits(2) AArch32.ReportedOuterAttrs(bits(2) attrs);
```

Library pseudocode for aarch32/dc/AArch32.DC

```
// AArch32.DC()
// ============
// Perform Data Cache Operation.

AArch32.DC(bits(32) regval, CacheOp cacheop, CacheOpScope opscope)
    CacheRecord cache;

    cache.acctype   = AccessType_DC;
    cache.cacheop   = cacheop;
    cache.opscope   = opscope;
    cache.cachetype = CacheType_Data;
    cache.security  = SecurityStateAtEL(PSTATE.EL);

    if opscope == CacheOpScope_SetWay then
        cache.shareability = Shareability_NSH;
        (cache.setnum, cache.waynum, cache.level) = DecodeSW(ZeroExtend(regval, 64),
                                                             CacheType_Data);

        if (cacheop == CacheOp_Invalidate && PSTATE.EL == EL1 && EL2Enabled() &&
            ((!ELUsingAArch32(EL2) && (HCR_EL2.SWIO == '1' || HCR_EL2.<DC,VM> != '00')) ||
              (ELUsingAArch32(EL2) && (HCR.SWIO == '1' || HCR.<DC,VM> != '00')))) then
            cache.cacheop = CacheOp_CleanInvalidate;
        CACHE_OP(cache);
        return;

    if EL2Enabled() then
        if PSTATE.EL IN {EL0, EL1} then
            cache.is_vmid_valid = TRUE;
            cache.vmid          = VMID[];
        else
            cache.is_vmid_valid = FALSE;
    else
        cache.is_vmid_valid = FALSE;

    if PSTATE.EL == EL0 then
        cache.is_asid_valid = TRUE;
        cache.asid          = ASID[];
    else
        cache.is_asid_valid = FALSE;

    need_translate = DCInstNeedsTranslation(opscope);
    vaddress = regval;

    size = 0;        // by default no watchpoint address
    if cacheop == CacheOp_Invalidate then
        size = integer IMPLEMENTATION_DEFINED "Data Cache Invalidate Watchpoint Size";
        assert size >= 4*(2^(UInt(CTR_EL0.DminLine))) && size <= 2048;
        assert UInt(size<32:0> AND (size-1)<32:0>) == 0; // size is power of 2
        vaddress = Align(regval, size);

    cache.translated = need_translate;
    cache.vaddress   = ZeroExtend(vaddress, 64);

    if need_translate then
        boolean aligned = TRUE;
        AccessDescriptor accdesc = CreateAccDescDC(cache);
        AddressDescriptor memaddrdesc = AArch32.TranslateAddress(vaddress, accdesc, aligned, size);
        if IsFault(memaddrdesc) then
            AArch32.Abort(regval, memaddrdesc.fault);

        cache.paddress = memaddrdesc.paddress;
        if opscope == CacheOpScope_PoC then
            cache.shareability = memaddrdesc.memattrs.shareability;
        else
            cache.shareability = Shareability_NSH;
    else
        cache.shareability = Shareability UNKNOWN;
        cache.paddress     = FullAddress UNKNOWN;

    if (cacheop == CacheOp_Invalidate && PSTATE.EL == EL1 && EL2Enabled() &&
```

```
            ((!ELUsingAArch32(EL2) && HCR_EL2.<DC,VM> != '00') ||
              (ELUsingAArch32(EL2) && HCR.<DC,VM> != '00'))) then
        cache.cacheop = CacheOp_CleanInvalidate;

    CACHE_OP(cache);
    return;
```

Library pseudocode for aarch32/debug/VCRMatch/AArch32.VCRMatch

```
// AArch32.VCRMatch()
// ==================

boolean AArch32.VCRMatch(bits(32) vaddress)

    boolean match;
    if UsingAArch32() && ELUsingAArch32(EL1) && PSTATE.EL != EL2 then
        // Each bit position in this string corresponds to a bit in DBGVCR and an exception vector.
        match_word = Zeros(32);

        ss = CurrentSecurityState();
        if vaddress<31:5> == ExcVectorBase()<31:5> then
            if HaveEL(EL3) && ss == SS_NonSecure then
                match_word<UInt(vaddress<4:2>) + 24> = '1';     // Non-secure vectors
            else
                match_word<UInt(vaddress<4:2>) + 0> = '1';      // Secure vectors (or no EL3)

        if (HaveEL(EL3) && ELUsingAArch32(EL3) && vaddress<31:5> == MVBAR<31:5> &&
            ss == SS_Secure) then
            match_word<UInt(vaddress<4:2>) + 8> = '1';          // Monitor vectors

        // Mask out bits not corresponding to vectors.
        bits(32) mask;
        if !HaveEL(EL3) then
            mask = '00000000':'00000000':'00000000':'11011110'; // DBGVCR[31:8] are RES0
        elsif !ELUsingAArch32(EL3) then
            mask = '11011110':'00000000':'00000000':'11011110'; // DBGVCR[15:8] are RES0
        else
            mask = '11011110':'00000000':'11011100':'11011110';

        match_word = match_word AND DBGVCR AND mask;
        match = !IsZero(match_word);

        // Check for UNPREDICTABLE case - match on Prefetch Abort and Data Abort vectors
        if !IsZero(match_word<28:27,12:11,4:3>) && DebugTarget() == PSTATE.EL then
            match = ConstrainUnpredictableBool(Unpredictable_VCMATCHDAPA);

        if !IsZero(vaddress<1:0>) && match then
            match = ConstrainUnpredictableBool(Unpredictable_VCMATCHHALF);
    else
        match = FALSE;

    return match;
```

Library pseudocode for aarch32/debug/authentication/AArch32.SelfHostedSecurePrivilegedInvasiveDebugEnabled

```
// AArch32.SelfHostedSecurePrivilegedInvasiveDebugEnabled()
// ========================================================

boolean AArch32.SelfHostedSecurePrivilegedInvasiveDebugEnabled()
    // The definition of this function is IMPLEMENTATION DEFINED.
    // In the recommended interface, AArch32.SelfHostedSecurePrivilegedInvasiveDebugEnabled returns
    // the state of the (DBGEN AND SPIDEN) signal.
    if !HaveEL(EL3) && NonSecureOnlyImplementation() then return FALSE;
    return DBGEN == Signal_High && SPIDEN == Signal_High;
```

Library pseudocode for aarch32/debug/breakpoint/AArch32.BreakpointMatch

```
// AArch32.BreakpointMatch()
// =========================
// Breakpoint matching in an AArch32 translation regime.

(boolean,boolean) AArch32.BreakpointMatch(integer n, bits(32) vaddress, AccessDescriptor accdesc,
                                          integer size)
    assert ELUsingAArch32(S1TranslationRegime());
    assert n < NumBreakpointsImplemented();

    enabled    = DBGBCR[n].E == '1';
    isbreakpnt = TRUE;
    linked     = DBGBCR[n].BT IN {'0x01'};
    linked_to  = FALSE;
    linked_n   = UInt(DBGBCR[n].LBN);

    state_match = AArch32.StateMatch(DBGBCR[n].SSC, DBGBCR[n].HMC, DBGBCR[n].PMC,
                                     linked, linked_n, isbreakpnt,  accdesc);
    (value_match, value_mismatch) = AArch32.BreakpointValueMatch(n, vaddress, linked_to);

    if size == 4 then                    // Check second halfword
        // If the breakpoint address and BAS of an Address breakpoint match the address of the
        // second halfword of an instruction, but not the address of the first halfword, it is
        // CONSTRAINED UNPREDICTABLE whether or not this breakpoint generates a Breakpoint debug
        // event.
        (match_i, mismatch_i) = AArch32.BreakpointValueMatch(n, vaddress + 2, linked_to);

        if !value_match && match_i then
            value_match = ConstrainUnpredictableBool(Unpredictable_BPMATCHHALF);

        if value_mismatch && !mismatch_i then
            value_mismatch = ConstrainUnpredictableBool(Unpredictable_BPMISMATCHHALF);

    if vaddress<1> == '1' && DBGBCR[n].BAS == '1111' then
        // The above notwithstanding, if DBGBCR[n].BAS == '1111', then it is CONSTRAINED
        // UNPREDICTABLE whether or not a Breakpoint debug event is generated for an instruction
        // at the address DBGBVR[n]+2.
        if value_match then
            value_match = ConstrainUnpredictableBool(Unpredictable_BPMATCHHALF);

        if !value_mismatch then
            value_mismatch = ConstrainUnpredictableBool(Unpredictable_BPMISMATCHHALF);

    match    = value_match && state_match && enabled;
    mismatch = value_mismatch && state_match && enabled;

    return (match, mismatch);
```

Library pseudocode for aarch32/debug/breakpoint/AArch32.BreakpointValueMatch

```
// AArch32.BreakpointValueMatch()
// ==============================
// The first result is whether an Address Match or Context breakpoint is programmed on the
// instruction at "address". The second result is whether an Address Mismatch breakpoint is
// programmed on the instruction, that is, whether the instruction should be stepped.

(boolean, boolean) AArch32.BreakpointValueMatch(integer n_in, bits(32) vaddress, boolean linked_to)

    // "n" is the identity of the breakpoint unit to match against.
    // "vaddress" is the current instruction address, ignored if linked_to is TRUE and for Context
    //   matching breakpoints.
    // "linked_to" is TRUE if this is a call from StateMatch for linking.
    integer n = n_in;
    Constraint c;

    // If a non-existent breakpoint then it is CONSTRAINED UNPREDICTABLE whether this gives
    // no match or the breakpoint is mapped to another UNKNOWN implemented breakpoint.
    if n >= NumBreakpointsImplemented() then
        (c, n) = ConstrainUnpredictableInteger(0, NumBreakpointsImplemented() - 1,
                                               Unpredictable_BPNOTIMPL);
        assert c IN {Constraint_DISABLED, Constraint_UNKNOWN};
        if c == Constraint_DISABLED then return (FALSE, FALSE);

    // If this breakpoint is not enabled, it cannot generate a match.
    // (This could also happen on a call from StateMatch for linking).
    if DBGBCR[n].E == '0' then return (FALSE, FALSE);

    dbgtype = DBGBCR[n].BT;

    (c, dbgtype) = AArch32.ReservedBreakpointType(n, dbgtype);
    if c == Constraint_DISABLED then return (FALSE, FALSE);
    // Otherwise the dbgtype value returned by AArch32.ReservedBreakpointType is valid.

    // Determine what to compare against.
    match_addr      = (dbgtype IN {'0x0x'});
    mismatch        = (dbgtype IN {'010x'});
    match_vmid      = (dbgtype IN {'10xx'});
    match_cid1      = (dbgtype IN {'xx1x'});
    match_cid2      = (dbgtype IN {'11xx'});
    linking_enabled = (dbgtype IN {'xxx1'});

    // If called from StateMatch, is is CONSTRAINED UNPREDICTABLE if the
    // breakpoint is not programmed with linking enabled.
    if linked_to && !linking_enabled then
        if !ConstrainUnpredictableBool(Unpredictable_BPLINKINGDISABLED) then
            return (FALSE, FALSE);

    // If called from BreakpointMatch return FALSE for Linked context ID and/or VMID matches.
    if !linked_to && linking_enabled && !match_addr then
        return (FALSE, FALSE);

    boolean bvr_match  = FALSE;
    boolean bxvr_match = FALSE;

    // Do the comparison.
    if match_addr then
        integer byte = UInt(vaddress<1:0>);
        assert byte IN {0,2};                     // "vaddress" is halfword aligned

        boolean byte_select_match = (DBGBCR[n].BAS<byte> == '1');
        integer top = 31;
        bvr_match = (vaddress<top:2> == DBGBVR[n]<top:2>) && byte_select_match;

    elsif match_cid1 then
        bvr_match = (PSTATE.EL != EL2 && CONTEXTIDR == DBGBVR[n]<31:0>);

    if match_vmid then
        bits(16) vmid;
        bits(16) bvr_vmid;
```

```
        if ELUsingAArch32(EL2) then
            vmid = ZeroExtend(VTTBR.VMID, 16);
            bvr_vmid = ZeroExtend(DBGBXVR[n]<7:0>, 16);
        elsif !Have16bitVMID() || VTCR_EL2.VS == '0' then
            vmid = ZeroExtend(VTTBR_EL2.VMID<7:0>, 16);
            bvr_vmid = ZeroExtend(DBGBXVR[n]<7:0>, 16);
        else
            vmid = VTTBR_EL2.VMID;
            bvr_vmid = DBGBXVR[n]<15:0>;

        bxvr_match = (PSTATE.EL IN {EL0, EL1} && EL2Enabled() && vmid == bvr_vmid);

    elsif match_cid2 then
        bxvr_match = (PSTATE.EL != EL3 && EL2Enabled() && !ELUsingAArch32(EL2) &&
                      DBGBXVR[n]<31:0> == CONTEXTIDR_EL2<31:0>);

    bvr_match_valid  = (match_addr || match_cid1);
    bxvr_match_valid = (match_vmid || match_cid2);

    match = (!bxvr_match_valid || bxvr_match) && (!bvr_match_valid || bvr_match);

    return (match && !mismatch, !match && mismatch);
```

Library pseudocode for aarch32/debug/breakpoint/AArch32.ReservedBreakpointType

```
// AArch32.ReservedBreakpointType()
// ================================
// Checks if the given DBGBCR<n>.BT value is reserved and will generate Constrained Unpredictable
// behavior, otherwise returns Constraint_NONE.

(Constraint, bits(4)) AArch32.ReservedBreakpointType(integer n, bits(4) bt_in)
    bits(4) bt       = bt_in;
    boolean reserved = FALSE;
    context_aware = n >= (NumBreakpointsImplemented() - NumContextAwareBreakpointsImplemented());

    // Address mismatch
    if bt IN {'010x'} && HaltOnBreakpointOrWatchpoint() then
        reserved = TRUE;

    // Context matching
    if !(bt IN {'0x0x'}) && !context_aware then
        reserved = TRUE;

    // EL2 extension
    if bt IN {'1xxx'} && !HaveEL(EL2) then
        reserved = TRUE;

    // Context matching
    if bt IN {'011x','11xx'} && !HaveVirtHostExt() && !HaveV82Debug() then
        reserved = TRUE;

    if reserved then
        Constraint c;
        (c, bt) = ConstrainUnpredictableBits(Unpredictable_RESBPTYPE, 4);
        assert c IN {Constraint_DISABLED, Constraint_UNKNOWN};
        if c == Constraint_DISABLED then
            return (c, bits(4) UNKNOWN);
        // Otherwise the value returned by ConstrainUnpredictableBits must be a not-reserved value

    return (Constraint_NONE, bt);
```

Library pseudocode for aarch32/debug/breakpoint/AArch32.StateMatch

```
// AArch32.StateMatch()
// ====================
// Determine whether a breakpoint or watchpoint is enabled in the current mode and state.

boolean AArch32.StateMatch(bits(2) ssc_in,  bit hmc_in, bits(2) pxc_in, boolean linked_in,
                           integer linked_n_in, boolean isbreakpnt, AccessDescriptor accdesc)

    // "ssc_in","hmc_in","pxc_in" are the control fields from the DBGBCR[n] or DBGWCR[n] register.
    // "linked_in" is TRUE if this is a linked breakpoint/watchpoint type.
    // "linked_n_in" is the linked breakpoint number from the DBGBCR[n] or DBGWCR[n] register.
    // "isbreakpnt" is TRUE for breakpoints, FALSE for watchpoints.
    // "accdesc" describes the properties of the access being matched.
    bit hmc          = hmc_in;
    bits(2) ssc      = ssc_in;
    bits(2) pxc      = pxc_in;
    boolean linked   = linked_in;
    integer linked_n = linked_n_in;

    // If parameters are set to a reserved type, behaves as either disabled or a defined type
    Constraint c;
    // SSCE value discarded as there is no SSCE bit in AArch32.
    (c, ssc, -, hmc, pxc) = CheckValidStateMatch(ssc, '0', hmc, pxc, isbreakpnt);
    if c == Constraint_DISABLED then return FALSE;
    // Otherwise the hmc,ssc,pxc values are either valid or the values returned by
    // CheckValidStateMatch are valid.

    pl2_match = HaveEL(EL2) && ((hmc == '1' && (ssc:pxc != '1000')) || ssc == '11');
    pl1_match = pxc<0> == '1';
    pl0_match = pxc<1> == '1';
    ssu_match = isbreakpnt && hmc == '0' && pxc == '00' && ssc != '11';

    boolean priv_match;
    if ssu_match then
        priv_match = PSTATE.M IN {M32_User,M32_Svc,M32_System};
    else
        case accdesc.el of
            when EL3 priv_match = pl1_match;           // EL3 and EL1 are both PL1
            when EL2 priv_match = pl2_match;
            when EL1 priv_match = pl1_match;
            when EL0 priv_match = pl0_match;

    // Security state match
    boolean ss_match;
    case ssc of
        when '00' ss_match = TRUE;                                     // Both
        when '01' ss_match = accdesc.ss == SS_NonSecure;               // Non-secure only
        when '10' ss_match = accdesc.ss == SS_Secure;                  // Secure only
        when '11' ss_match = (hmc == '1' || accdesc.ss == SS_Secure);  // HMC=1 -> Both,
                                                                       // HMC=0 -> Secure only

    boolean linked_match = FALSE;

    if linked then
        // "linked_n" must be an enabled context-aware breakpoint unit.
        // If it is not context-aware then it is CONSTRAINED UNPREDICTABLE whether
        // this gives no match, gives a match without linking, or linked_n is mapped to some
        // UNKNOWN breakpoint that is context-aware.
        if !IsContextMatchingBreakpoint(linked_n) then
            (first_ctx_cmp, last_ctx_cmp) = ContextMatchingBreakpointRange();
            (c, linked_n) = ConstrainUnpredictableInteger(first_ctx_cmp, last_ctx_cmp,
                                                          Unpredictable_BPNOTCTXCMP);
            assert c IN {Constraint_DISABLED, Constraint_NONE, Constraint_UNKNOWN};

            case c of
                when Constraint_DISABLED  return FALSE;      // Disabled
                when Constraint_NONE      linked = FALSE;    // No linking
                // Otherwise ConstrainUnpredictableInteger returned a context-aware breakpoint

        vaddress  = bits(32) UNKNOWN;
        linked_to = TRUE;
```

```
        (linked_match,-) = AArch32.BreakpointValueMatch(linked_n, vaddress, linked_to);

    return priv_match && ss_match && (!linked || linked_match);
```

Library pseudocode for aarch32/debug/enables/AArch32.GenerateDebugExceptions

```
// AArch32.GenerateDebugExceptions()
// =================================

boolean AArch32.GenerateDebugExceptions()
    ss = CurrentSecurityState();
    return AArch32.GenerateDebugExceptionsFrom(PSTATE.EL, ss);
```

Library pseudocode for aarch32/debug/enables/AArch32.GenerateDebugExceptionsFrom

```
// AArch32.GenerateDebugExceptionsFrom()
// =====================================

boolean AArch32.GenerateDebugExceptionsFrom(bits(2) from_el, SecurityState from_state)

    if !ELUsingAArch32(DebugTargetFrom(from_state)) then
        mask = '0';    // No PSTATE.D in AArch32 state
        return AArch64.GenerateDebugExceptionsFrom(from_el, from_state, mask);

    if DBGOSLSR.OSLK == '1' || DoubleLockStatus() || Halted() then
        return FALSE;

    boolean enabled;
    if HaveEL(EL3) && from_state == SS_Secure then
        assert from_el != EL2;    // Secure EL2 always uses AArch64
        if IsSecureEL2Enabled() then
            // Implies that EL3 and EL2 both using AArch64
            enabled = MDCR_EL3.SDD == '0';
        else
            spd = if ELUsingAArch32(EL3) then SDCR.SPD else MDCR_EL3.SPD32;
            if spd<1> == '1' then
                enabled = spd<0> == '1';
            else
                // SPD == 0b01 is reserved, but behaves the same as 0b00.
                enabled = AArch32.SelfHostedSecurePrivilegedInvasiveDebugEnabled();
        if from_el == EL0 then enabled = enabled || SDER.SUIDEN == '1';
    else
        enabled = from_el != EL2;

    return enabled;
```

Library pseudocode for aarch32/debug/pmu/AArch32.ClearEventCounters

```
// AArch32.ClearEventCounters()
// ============================
// Zero all the event counters.

AArch32.ClearEventCounters()
    if HaveAArch64() then
        // Force the counter to be cleared as a 64-bit counter.
        AArch64.ClearEventCounters();
        return;

    integer counters = AArch32.GetNumEventCountersAccessible();
    if counters != 0 then
        for idx = 0 to counters - 1
            PMEVCNTR[idx] = Zeros(32);
```

Library pseudocode for aarch32/debug/pmu/AArch32.GetNumEventCountersAccessible

```
// AArch32.GetNumEventCountersAccessible()
// =======================================
// Return the number of event counters that can be accessed at the current Exception level.

integer AArch32.GetNumEventCountersAccessible()
    integer n;
    integer total_counters = GetNumEventCounters();
    // Software can reserve some counters for EL2
    if PSTATE.EL IN {EL1, EL0} && EL2Enabled() then
        n = UInt(if !ELUsingAArch32(EL2) then MDCR_EL2.HPMN else HDCR.HPMN);
        if n > total_counters || (!HaveFeatHPMN0() && n == 0) then
            (-, n) = ConstrainUnpredictableInteger(0, total_counters,
                                                   Unpredictable_PMUEVENTCOUNTER);
    else
        n = total_counters;

    return n;
```

Library pseudocode for aarch32/debug/pmu/AArch32.IncrementCycleCounter

```
// AArch32.IncrementCycleCounter()
// ===============================
// Increment the cycle counter and possibly set overflow bits.

AArch32.IncrementCycleCounter()
    if (CountPMUEvents(CYCLE_COUNTER_ID) &&
        (PMCR.LC == '1' || PMCR.D == '0' || HasElapsed64Cycles())) then
        integer old_value = UInt(PMCCNTR);
        integer new_value = old_value + 1;
        PMCCNTR = new_value<63:0>;

        integer ovflw = if PMCR.LC == '1' then 64 else 32;

        if old_value<64:ovflw> != new_value<64:ovflw> then
            PMOVSSET.C = '1';
            PMOVSR.C = '1';
```

Library pseudocode for aarch32/debug/pmu/AArch32.IncrementEventCounter

```
// AArch32.IncrementEventCounter()
// ===============================
// Increment the specified event counter by the specified amount.

AArch32.IncrementEventCounter(integer idx, integer increment)
    if HaveAArch64() then
        // Force the counter to be incremented as a 64-bit counter.
        AArch64.IncrementEventCounter(idx, increment);
        return;

    // In this model, event counters in an AArch32-only implementation are 32 bits and
    // the LP bits are RES0 in this model, even if FEAT_PMUv3p5 is implemented.
    integer old_value;
    integer new_value;
    integer ovflw;

    old_value = UInt(PMEVCNTR[idx]);
    new_value = old_value + PMUCountValue(idx, increment);

    PMEVCNTR[idx] = new_value<31:0>;
    ovflw = 32;

    if old_value<64:ovflw> != new_value<64:ovflw> then
        PMOVSSET<idx> = '1';
        PMOVSR<idx> = '1';
        // Check for the CHAIN event from an even counter
        if idx<0> == '0' && idx + 1 < GetNumEventCounters() then
            PMUEvent(PMU_EVENT_CHAIN, 1, idx + 1);
```

Library pseudocode for aarch32/debug/pmu/AArch32.PMUCycle

```
// AArch32.PMUCycle()
// ==================
// Called at the end of each cycle to increment event counters and
// check for PMU overflow. In pseudocode, a cycle ends after the
// execution of the operational pseudocode.

AArch32.PMUCycle()
    if HaveAArch64() then
        AArch64.PMUCycle();
        return;

    if !HavePMUv3() then
        return;

    PMUEvent(PMU_EVENT_CPU_CYCLES);

    integer counters = GetNumEventCounters();
    if counters != 0 then
        for idx = 0 to counters - 1
            if CountPMUEvents(idx) then
                integer accumulated = PMUEventAccumulator[idx];
                AArch32.IncrementEventCounter(idx, accumulated);
            PMUEventAccumulator[idx] = 0;

    AArch32.IncrementCycleCounter();
    CheckForPMUOverflow();
```

Library pseudocode for aarch32/debug/pmu/AArch32.PMUSwIncrement

```
// AArch32.PMUSwIncrement()
// ========================
// Generate PMU Events on a write to PMSWINC.

AArch32.PMUSwIncrement(bits(32) sw_incr)
    integer counters = AArch32.GetNumEventCountersAccessible();
    if counters != 0 then
        for idx = 0 to counters - 1
            if sw_incr<idx> == '1' then
                PMUEvent(PMU_EVENT_SW_INCR, 1, idx);
```

Library pseudocode for aarch32/debug/takeexceptiondbg/AArch32.EnterHypModeInDebugState

```
// AArch32.EnterHypModeInDebugState()
// ==================================
// Take an exception in Debug state to Hyp mode.

AArch32.EnterHypModeInDebugState(ExceptionRecord except)
    SynchronizeContext();
    assert HaveEL(EL2) && CurrentSecurityState() == SS_NonSecure && ELUsingAArch32(EL2);

    AArch32.ReportHypEntry(except);
    AArch32.WriteMode(M32_Hyp);
    SPSR[] = bits(32) UNKNOWN;
    ELR_hyp = bits(32) UNKNOWN;
    // In Debug state, the PE always execute T32 instructions when in AArch32 state, and
    // PSTATE.{SS,A,I,F} are not observable so behave as UNKNOWN.
    PSTATE.T = '1';                             // PSTATE.J is RES0
    PSTATE.<SS,A,I,F> = bits(4) UNKNOWN;
    DLR = bits(32) UNKNOWN;
    DSPSR = bits(32) UNKNOWN;
    PSTATE.E = HSCTLR.EE;
    PSTATE.IL = '0';
    PSTATE.IT = '00000000';
    if HaveSSBSExt() then PSTATE.SSBS = bit UNKNOWN;
    EDSCR.ERR = '1';
    UpdateEDSCRFields();

    EndOfInstruction();
```

Library pseudocode for aarch32/debug/takeexceptiondbg/AArch32.EnterModeInDebugState

```
// AArch32.EnterModeInDebugState()
// ===============================
// Take an exception in Debug state to a mode other than Monitor and Hyp mode.

AArch32.EnterModeInDebugState(bits(5) target_mode)
    SynchronizeContext();
    assert ELUsingAArch32(EL1) && PSTATE.EL != EL2;

    if PSTATE.M == M32_Monitor then SCR.NS = '0';
    AArch32.WriteMode(target_mode);
    SPSR[] = bits(32) UNKNOWN;
    R[14] = bits(32) UNKNOWN;
    // In Debug state, the PE always execute T32 instructions when in AArch32 state, and
    // PSTATE.{SS,A,I,F} are not observable so behave as UNKNOWN.
    PSTATE.T = '1';                             // PSTATE.J is RES0
    PSTATE.<SS,A,I,F> = bits(4) UNKNOWN;
    DLR = bits(32) UNKNOWN;
    DSPSR = bits(32) UNKNOWN;
    PSTATE.E = SCTLR.EE;
    PSTATE.IL = '0';
    PSTATE.IT = '00000000';
    if HavePANExt() && SCTLR.SPAN == '0' then PSTATE.PAN = '1';
    if HaveSSBSExt() then PSTATE.SSBS = bit UNKNOWN;
    EDSCR.ERR = '1';
    UpdateEDSCRFields();                        // Update EDSCR processor state flags.

    EndOfInstruction();
```

Library pseudocode for aarch32/debug/takeexceptiondbg/AArch32.EnterMonitorModeInDebugState

```
// AArch32.EnterMonitorModeInDebugState()
// ======================================
// Take an exception in Debug state to Monitor mode.

AArch32.EnterMonitorModeInDebugState()
    SynchronizeContext();
    assert HaveEL(EL3) && ELUsingAArch32(EL3);
    from_secure = CurrentSecurityState() == SS_Secure;
    if PSTATE.M == M32_Monitor then SCR.NS = '0';
    AArch32.WriteMode(M32_Monitor);
    SPSR[] = bits(32) UNKNOWN;
    R[14] = bits(32) UNKNOWN;
    // In Debug state, the PE always execute T32 instructions when in AArch32 state, and
    // PSTATE.{SS,A,I,F} are not observable so behave as UNKNOWN.
    PSTATE.T = '1';                             // PSTATE.J is RES0
    PSTATE.<SS,A,I,F> = bits(4) UNKNOWN;
    PSTATE.E = SCTLR.EE;
    PSTATE.IL = '0';
    PSTATE.IT = '00000000';
    if HavePANExt() then
        if !from_secure then
            PSTATE.PAN = '0';
        elsif SCTLR.SPAN == '0' then
            PSTATE.PAN = '1';
    if HaveSSBSExt() then PSTATE.SSBS = bit UNKNOWN;
    DLR = bits(32) UNKNOWN;
    DSPSR = bits(32) UNKNOWN;
    EDSCR.ERR = '1';
    UpdateEDSCRFields();                        // Update EDSCR processor state flags.

    EndOfInstruction();
```

Library pseudocode for aarch32/debug/watchpoint/AArch32.WatchpointByteMatch

```
// AArch32.WatchpointByteMatch()
// =============================

boolean AArch32.WatchpointByteMatch(integer n, bits(32) vaddress)

    integer top = 31;
    bottom = if DBGWVR[n]<2> == '1' then 2 else 3;            // Word or doubleword
    byte_select_match = (DBGWCR[n].BAS<UInt(vaddress<bottom-1:0>)> != '0');
    mask = UInt(DBGWCR[n].MASK);

    // If DBGWCR[n].MASK is a nonzero value and DBGWCR[n].BAS is not set to '11111111', or
    // DBGWCR[n].BAS specifies a non-contiguous set of bytes behavior is CONSTRAINED
    // UNPREDICTABLE.
    if mask > 0 && !IsOnes(DBGWCR[n].BAS) then
        byte_select_match = ConstrainUnpredictableBool(Unpredictable_WPMASKANDBAS);
    else
        LSB = (DBGWCR[n].BAS AND NOT(DBGWCR[n].BAS - 1));  MSB = (DBGWCR[n].BAS + LSB);
        if !IsZero(MSB AND (MSB - 1)) then                     // Not contiguous
            byte_select_match = ConstrainUnpredictableBool(Unpredictable_WPBASCONTIGUOUS);
            bottom = 3;                                        // For the whole doubleword

    // If the address mask is set to a reserved value, the behavior is CONSTRAINED UNPREDICTABLE.
    if mask > 0 && mask <= 2 then
        Constraint c;
        (c, mask) = ConstrainUnpredictableInteger(3, 31, Unpredictable_RESWPMASK);
        assert c IN {Constraint_DISABLED, Constraint_NONE, Constraint_UNKNOWN};
        case c of
            when Constraint_DISABLED  return FALSE;            // Disabled
            when Constraint_NONE      mask = 0;                // No masking
            // Otherwise the value returned by ConstrainUnpredictableInteger is a not-reserved value

    boolean WVR_match;
    if mask > bottom then
        WVR_match = (vaddress<top:mask> == DBGWVR[n]<top:mask>);
        // If masked bits of DBGWVR_EL1[n] are not zero, the behavior is CONSTRAINED UNPREDICTABLE.
        if WVR_match && !IsZero(DBGWVR[n]<mask-1:bottom>) then
            WVR_match = ConstrainUnpredictableBool(Unpredictable_WPMASKEDBITS);
    else
        WVR_match = vaddress<top:bottom> == DBGWVR[n]<top:bottom>;

    return WVR_match && byte_select_match;
```

Library pseudocode for aarch32/debug/watchpoint/AArch32.WatchpointMatch

```
// AArch32.WatchpointMatch()
// =========================
// Watchpoint matching in an AArch32 translation regime.

boolean AArch32.WatchpointMatch(integer n, bits(32) vaddress, integer size,
                                AccessDescriptor accdesc)
    assert ELUsingAArch32(S1TranslationRegime());
    assert n < NumWatchpointsImplemented();

    enabled = DBGWCR[n].E == '1';
    linked = DBGWCR[n].WT == '1';
    isbreakpnt = FALSE;
    linked_n = UInt(DBGWCR_EL1[n].LBN);
    state_match = AArch32.StateMatch(DBGWCR[n].SSC, DBGWCR[n].HMC, DBGWCR[n].PAC,
                                     linked, linked_n, isbreakpnt, accdesc);

    boolean ls_match;
    case DBGWCR[n].LSC<1:0> of
        when '00' ls_match = FALSE;
        when '01' ls_match = accdesc.read;
        when '10' ls_match = accdesc.write || accdesc.acctype == AccessType_DC;
        when '11' ls_match = TRUE;

    value_match = FALSE;
    for byte = 0 to size - 1
        value_match = value_match || AArch32.WatchpointByteMatch(n, vaddress + byte);

    return value_match && state_match && ls_match && enabled;
```

Library pseudocode for aarch32/exceptions/aborts/AArch32.Abort

```
// AArch32.Abort()
// ===============
// Abort and Debug exception handling in an AArch32 translation regime.

AArch32.Abort(bits(32) vaddress, FaultRecord fault)

    // Check if routed to AArch64 state
    route_to_aarch64 = PSTATE.EL == EL0 && !ELUsingAArch32(EL1);

    if !route_to_aarch64 && EL2Enabled() && !ELUsingAArch32(EL2) then
        route_to_aarch64 = (HCR_EL2.TGE == '1' || IsSecondStage(fault) ||
                            (HaveRASExt() && HCR_EL2.TEA == '1' && IsExternalAbort(fault)) ||
                            (IsDebugException(fault) && MDCR_EL2.TDE == '1'));

    if !route_to_aarch64 && HaveEL(EL3) && !ELUsingAArch32(EL3) then
        route_to_aarch64 = EffectiveEA() == '1' && IsExternalAbort(fault);

    if route_to_aarch64 then
        AArch64.Abort(ZeroExtend(vaddress,  64), fault);
    elsif fault.access.acctype == AccessType_IFETCH then
        AArch32.TakePrefetchAbortException(vaddress, fault);
    else
        AArch32.TakeDataAbortException(vaddress, fault);
```

Library pseudocode for aarch32/exceptions/aborts/AArch32.AbortSyndrome

```
// AArch32.AbortSyndrome()
// =======================
// Creates an exception syndrome record for Abort  exceptions
// taken to Hyp mode
// from an AArch32 translation regime.

ExceptionRecord AArch32.AbortSyndrome(Exception exceptype, FaultRecord fault,
                                      bits(32) vaddress, bits(2) target_el)
    except = ExceptionSyndrome(exceptype);

    d_side = exceptype == Exception_DataAbort;

    except.syndrome = AArch32.FaultSyndrome(d_side, fault);
    except.vaddress = ZeroExtend(vaddress, 64);

    if IPAValid(fault) then
        except.ipavalid = TRUE;
        except.NS = if fault.ipaddress.paspace == PAS_NonSecure then '1' else '0';
        except.ipaddress = ZeroExtend(fault.ipaddress.address,  56);
    else
        except.ipavalid = FALSE;

    return except;
```

Library pseudocode for aarch32/exceptions/aborts/AArch32.CheckPCAlignment

```
// AArch32.CheckPCAlignment()
// ==========================

AArch32.CheckPCAlignment()
    bits(32) pc = ThisInstrAddr(32);

    if (CurrentInstrSet() == InstrSet_A32 && pc<1> == '1') || pc<0> == '1' then
        if AArch32.GeneralExceptionsToAArch64() then AArch64.PCAlignmentFault();

        AccessDescriptor accdesc = CreateAccDescIFetch();
        FaultRecord fault = NoFault(accdesc);
        // Generate an Alignment fault Prefetch Abort exception
        fault.statuscode = Fault_Alignment;
        AArch32.Abort(pc, fault);
```

Library pseudocode for aarch32/exceptions/aborts/AArch32.CommonFaultStatus

```
// AArch32.CommonFaultStatus()
// ===========================
// Return the common part of the fault status on reporting a Data
// or Prefetch Abort.

bits(32) AArch32.CommonFaultStatus(FaultRecord fault, boolean long_format)
    bits(32) target = Zeros(32);
    if HaveRASExt() && IsAsyncAbort(fault) then
        ErrorState errstate = AArch32.PEErrorState(fault);
        target<15:14> = AArch32.EncodeAsyncErrorSyndrome(errstate);   // AET
    if IsExternalAbort(fault) then target<12> = fault.extflag;        // ExT
    target<9> = if long_format then '1' else '0';                     // LPAE
    if long_format then                                               // Long-descriptor format
        target<5:0>    = EncodeLDFSC(fault.statuscode, fault.level);  // STATUS
    else                                                              // Short-descriptor format
        target<10,3:0> = EncodeSDFSC(fault.statuscode, fault.level);  // FS
    return target;
```

Library pseudocode for aarch32/exceptions/aborts/AArch32.ReportDataAbort

```
// AArch32.ReportDataAbort()
// =========================
// Report syndrome information for aborts taken to modes other than Hyp mode.

AArch32.ReportDataAbort(boolean route_to_monitor, FaultRecord fault,
                        bits(32) vaddress)
    boolean long_format;
    if route_to_monitor && CurrentSecurityState() != SS_Secure then
        long_format = ((TTBCR_S.EAE == '1') ||
                       (IsExternalSyncAbort(fault) && ((PSTATE.EL == EL2 || TTBCR.EAE == '1') ||
                        (fault.secondstage && (boolean IMPLEMENTATION_DEFINED
                                               "Report abort using Long-descriptor format")))));
    else
        long_format = TTBCR.EAE == '1';
    bits(32) syndrome = AArch32.CommonFaultStatus(fault, long_format);

    // bits of syndrome that are not common to I and D side
    if fault.access.acctype IN {AccessType_DC, AccessType_IC, AccessType_AT} then
        syndrome<13> = '1';                              // CM
        syndrome<11> = '1';                              // WnR
    else
        syndrome<11> = if fault.write then '1' else '0'; // WnR

    if !long_format then
        syndrome<7:4> = fault.domain;                    // Domain

    if fault.access.acctype == AccessType_IC then
        bits(32) i_syndrome;
        if (!long_format &&
            boolean IMPLEMENTATION_DEFINED "Report I-cache maintenance fault in IFSR") then
            i_syndrome = syndrome;
            syndrome<10,3:0> = EncodeSDFSC(Fault_ICacheMaint, 1);
        else
            i_syndrome = bits(32) UNKNOWN;
        if route_to_monitor then
            IFSR_S = i_syndrome;
        else
            IFSR = i_syndrome;

    if route_to_monitor then
        DFSR_S = syndrome;
        DFAR_S = vaddress;
    else
        DFSR = syndrome;
        DFAR = vaddress;

    return;
```

Library pseudocode for aarch32/exceptions/aborts/AArch32.ReportPrefetchAbort

```
// AArch32.ReportPrefetchAbort()
// =============================
// Report syndrome information for aborts taken to modes other than Hyp mode.

AArch32.ReportPrefetchAbort(boolean route_to_monitor, FaultRecord fault, bits(32) vaddress)
    // The encoding used in the IFSR can be Long-descriptor format or Short-descriptor format.
    // Normally, the current translation table format determines the format. For an abort from
    // Non-secure state to Monitor mode, the IFSR uses the Long-descriptor format if any of the
    // following applies:
    // * The Secure TTBCR.EAE is set to 1.
    // * It is taken from Hyp mode.
    // * It is taken from EL1 or EL0, and the Non-secure TTBCR.EAE is set to 1.
    long_format = FALSE;
    if route_to_monitor && CurrentSecurityState() != SS_Secure then
        long_format = TTBCR_S.EAE == '1' || PSTATE.EL == EL2 || TTBCR.EAE == '1';
    else
        long_format = TTBCR.EAE == '1';

    bits(32) fsr = AArch32.CommonFaultStatus(fault, long_format);

    if route_to_monitor then
        IFSR_S = fsr;
        IFAR_S = vaddress;
    else
        IFSR = fsr;
        IFAR = vaddress;

    return;
```

Library pseudocode for aarch32/exceptions/aborts/AArch32.TakeDataAbortException

```
// AArch32.TakeDataAbortException()
// ================================

AArch32.TakeDataAbortException(bits(32) vaddress, FaultRecord fault)
    route_to_monitor = HaveEL(EL3) && EffectiveEA() == '1' && IsExternalAbort(fault);
    route_to_hyp = (EL2Enabled() && PSTATE.EL IN {EL0, EL1} &&
                    (HCR.TGE == '1' ||
                     (HaveRASExt() && HCR2.TEA == '1' && IsExternalAbort(fault)) ||
                     (IsDebugException(fault) && HDCR.TDE == '1') ||
                     IsSecondStage(fault)));

    bits(32) preferred_exception_return = ThisInstrAddr(32);
    vect_offset = 0x10;
    lr_offset = 8;

    if IsDebugException(fault) then DBGDSCRext.MOE = fault.debugmoe;
    if route_to_monitor then
        AArch32.ReportDataAbort(route_to_monitor, fault, vaddress);
        AArch32.EnterMonitorMode(preferred_exception_return, lr_offset, vect_offset);
    elsif PSTATE.EL == EL2 || route_to_hyp then
        except = AArch32.AbortSyndrome(Exception_DataAbort, fault, vaddress, EL2);
        if PSTATE.EL == EL2 then
            AArch32.EnterHypMode(except, preferred_exception_return, vect_offset);
        else
            AArch32.EnterHypMode(except, preferred_exception_return, 0x14);
    else
        AArch32.ReportDataAbort(route_to_monitor, fault, vaddress);
        AArch32.EnterMode(M32_Abort, preferred_exception_return, lr_offset, vect_offset);
```

Library pseudocode for aarch32/exceptions/aborts/AArch32.TakePrefetchAbortException

```
// AArch32.TakePrefetchAbortException()
// ====================================

AArch32.TakePrefetchAbortException(bits(32) vaddress, FaultRecord fault)
    route_to_monitor = HaveEL(EL3) && EffectiveEA() == '1' && IsExternalAbort(fault);

    route_to_hyp = (EL2Enabled() && PSTATE.EL IN {EL0, EL1} &&
                    (HCR.TGE == '1' ||
                     (HaveRASExt() && HCR2.TEA == '1' && IsExternalAbort(fault)) ||
                     (IsDebugException(fault) && HDCR.TDE == '1') ||
                     IsSecondStage(fault)));

    ExceptionRecord except;
    bits(32) preferred_exception_return = ThisInstrAddr(32);
    vect_offset = 0x0C;
    lr_offset = 4;

    if IsDebugException(fault) then DBGDSCRext.MOE = fault.debugmoe;
    if route_to_monitor then
        AArch32.ReportPrefetchAbort(route_to_monitor, fault, vaddress);
        AArch32.EnterMonitorMode(preferred_exception_return, lr_offset, vect_offset);
    elsif PSTATE.EL == EL2 || route_to_hyp then
        if fault.statuscode == Fault_Alignment then             // PC Alignment fault
            except = ExceptionSyndrome(Exception_PCAlignment);
            except.vaddress = ThisInstrAddr(64);
        else
            except = AArch32.AbortSyndrome(Exception_InstructionAbort, fault, vaddress, EL2);
        if PSTATE.EL == EL2 then
            AArch32.EnterHypMode(except, preferred_exception_return, vect_offset);
        else
            AArch32.EnterHypMode(except, preferred_exception_return, 0x14);
    else
        AArch32.ReportPrefetchAbort(route_to_monitor, fault, vaddress);
        AArch32.EnterMode(M32_Abort, preferred_exception_return, lr_offset, vect_offset);
```

Library pseudocode for aarch32/exceptions/async/AArch32.TakePhysicalFIQException

```
// AArch32.TakePhysicalFIQException()
// ==================================

AArch32.TakePhysicalFIQException()

    // Check if routed to AArch64 state
    route_to_aarch64 = PSTATE.EL == EL0 && !ELUsingAArch32(EL1);
    if !route_to_aarch64 && EL2Enabled() && !ELUsingAArch32(EL2) then
        route_to_aarch64 = HCR_EL2.TGE == '1' || (HCR_EL2.FMO == '1' && !IsInHost());

    if !route_to_aarch64 && HaveEL(EL3) && !ELUsingAArch32(EL3) then
        route_to_aarch64 = SCR_EL3.FIQ == '1';

    if route_to_aarch64 then AArch64.TakePhysicalFIQException();
    route_to_monitor = HaveEL(EL3) && SCR.FIQ == '1';
    route_to_hyp = (PSTATE.EL IN {EL0, EL1} && EL2Enabled() &&
                    (HCR.TGE == '1' || HCR.FMO == '1'));
    bits(32) preferred_exception_return = ThisInstrAddr(32);
    vect_offset = 0x1C;
    lr_offset = 4;
    if route_to_monitor then
        AArch32.EnterMonitorMode(preferred_exception_return, lr_offset, vect_offset);
    elsif PSTATE.EL == EL2 || route_to_hyp then
        except = ExceptionSyndrome(Exception_FIQ);
        AArch32.EnterHypMode(except, preferred_exception_return, vect_offset);
    else
        AArch32.EnterMode(M32_FIQ, preferred_exception_return, lr_offset, vect_offset);
```

Library pseudocode for aarch32/exceptions/async/AArch32.TakePhysicalIRQException

```
// AArch32.TakePhysicalIRQException()
// ==================================
// Take an enabled physical IRQ exception.

AArch32.TakePhysicalIRQException()

    // Check if routed to AArch64 state
    route_to_aarch64 = PSTATE.EL == EL0 && !ELUsingAArch32(EL1);
    if !route_to_aarch64 && EL2Enabled() && !ELUsingAArch32(EL2) then
        route_to_aarch64 = HCR_EL2.TGE == '1' || (HCR_EL2.IMO == '1' && !IsInHost());
    if !route_to_aarch64 && HaveEL(EL3) && !ELUsingAArch32(EL3) then
        route_to_aarch64 = SCR_EL3.IRQ == '1';

    if route_to_aarch64 then AArch64.TakePhysicalIRQException();

    route_to_monitor = HaveEL(EL3) && SCR.IRQ == '1';
    route_to_hyp = (PSTATE.EL IN {EL0, EL1} && EL2Enabled() &&
                    (HCR.TGE == '1' || HCR.IMO == '1'));
    bits(32) preferred_exception_return = ThisInstrAddr(32);
    vect_offset = 0x18;
    lr_offset = 4;
    if route_to_monitor then
        AArch32.EnterMonitorMode(preferred_exception_return, lr_offset, vect_offset);
    elsif PSTATE.EL == EL2 || route_to_hyp then
        except = ExceptionSyndrome(Exception_IRQ);
        AArch32.EnterHypMode(except, preferred_exception_return, vect_offset);
    else
        AArch32.EnterMode(M32_IRQ, preferred_exception_return, lr_offset, vect_offset);
```

Library pseudocode for aarch32/exceptions/async/AArch32.TakePhysicalSErrorException

```
// AArch32.TakePhysicalSErrorException()
// =====================================

AArch32.TakePhysicalSErrorException(boolean implicit_esb)
    // Check if routed to AArch64 state
    route_to_aarch64 = PSTATE.EL == EL0 && !ELUsingAArch32(EL1);

    if !route_to_aarch64 && EL2Enabled() && !ELUsingAArch32(EL2) then
        route_to_aarch64 = (HCR_EL2.TGE == '1' || (!IsInHost() && HCR_EL2.AMO == '1'));
    if !route_to_aarch64 && HaveEL(EL3) && !ELUsingAArch32(EL3) then
        route_to_aarch64 = EffectiveEA() == '1';

    if route_to_aarch64 then
        AArch64.TakePhysicalSErrorException(implicit_esb);

    route_to_monitor = HaveEL(EL3) && SCR.EA == '1';
    route_to_hyp = (PSTATE.EL IN {EL0, EL1} && EL2Enabled() &&
                    (HCR.TGE == '1' || HCR.AMO == '1'));
    bits(32) preferred_exception_return = ThisInstrAddr(32);
    vect_offset = 0x10;
    lr_offset = 8;

    bits(2) target_el;
    if route_to_monitor then
        target_el = EL3;
    elsif PSTATE.EL == EL2 || route_to_hyp then
        target_el = EL2;
    else
        target_el = EL1;

    FaultRecord fault = GetPendingPhysicalSError();
    vaddress = bits(32) UNKNOWN;
    except = AArch32.AbortSyndrome(Exception_DataAbort, fault, vaddress, target_el);

    if IsSErrorEdgeTriggered() then
        ClearPendingPhysicalSError();
    case target_el of
        when EL3
            AArch32.ReportDataAbort(route_to_monitor, fault, vaddress);
            AArch32.EnterMonitorMode(preferred_exception_return, lr_offset, vect_offset);
        when EL2
            if PSTATE.EL == EL2 then
                AArch32.EnterHypMode(except, preferred_exception_return, vect_offset);
            else
                AArch32.EnterHypMode(except, preferred_exception_return, 0x14);
        when EL1
            AArch32.ReportDataAbort(route_to_monitor, fault, vaddress);
            AArch32.EnterMode(M32_Abort, preferred_exception_return, lr_offset, vect_offset);
        otherwise
            Unreachable();
```

Library pseudocode for aarch32/exceptions/async/AArch32.TakeVirtualFIQException

```
// AArch32.TakeVirtualFIQException()
// =================================

AArch32.TakeVirtualFIQException()
    assert PSTATE.EL IN {EL0, EL1} && EL2Enabled();
    if ELUsingAArch32(EL2) then  // Virtual IRQ enabled if TGE==0 and FMO==1
        assert HCR.TGE == '0' && HCR.FMO == '1';
    else
        assert HCR_EL2.TGE == '0' && HCR_EL2.FMO == '1';
    // Check if routed to AArch64 state
    if PSTATE.EL == EL0 && !ELUsingAArch32(EL1) then AArch64.TakeVirtualFIQException();

    bits(32) preferred_exception_return = ThisInstrAddr(32);
    vect_offset = 0x1C;
    lr_offset = 4;

    AArch32.EnterMode(M32_FIQ, preferred_exception_return, lr_offset, vect_offset);
```

Library pseudocode for aarch32/exceptions/async/AArch32.TakeVirtualIRQException

```
// AArch32.TakeVirtualIRQException()
// =================================

AArch32.TakeVirtualIRQException()
    assert PSTATE.EL IN {EL0, EL1} && EL2Enabled();

    if ELUsingAArch32(EL2) then   // Virtual IRQs enabled if TGE==0 and IMO==1
        assert HCR.TGE == '0' && HCR.IMO == '1';
    else
        assert HCR_EL2.TGE == '0' && HCR_EL2.IMO == '1';

    // Check if routed to AArch64 state
    if PSTATE.EL == EL0 && !ELUsingAArch32(EL1) then AArch64.TakeVirtualIRQException();

    bits(32) preferred_exception_return = ThisInstrAddr(32);
    vect_offset = 0x18;
    lr_offset = 4;

    AArch32.EnterMode(M32_IRQ, preferred_exception_return, lr_offset, vect_offset);
```

Library pseudocode for aarch32/exceptions/async/AArch32.TakeVirtualSErrorException

```
// AArch32.TakeVirtualSErrorException()
// ====================================

AArch32.TakeVirtualSErrorException()

    assert PSTATE.EL IN {EL0, EL1} && EL2Enabled();
    if ELUsingAArch32(EL2) then   // Virtual SError enabled if TGE==0 and AMO==1
        assert HCR.TGE == '0' && HCR.AMO == '1';
    else
        assert HCR_EL2.TGE == '0' && HCR_EL2.AMO == '1';
    // Check if routed to AArch64 state
    if PSTATE.EL == EL0 && !ELUsingAArch32(EL1) then AArch64.TakeVirtualSErrorException();
    route_to_monitor = FALSE;

    bits(32) preferred_exception_return = ThisInstrAddr(32);
    vect_offset = 0x10;
    lr_offset = 8;

    vaddress = bits(32) UNKNOWN;
    parity = FALSE;
    Fault fault = Fault_AsyncExternal;
    integer level = integer UNKNOWN;
    bits(32) fsr = Zeros(32);
    if HaveRASExt() then
        if ELUsingAArch32(EL2) then
            fsr<15:14> =  VDFSR.AET;
            fsr<12> =  VDFSR.ExT;
        else
            fsr<15:14> =  VSESR_EL2.AET;
            fsr<12> =  VSESR_EL2.ExT;
    else
        fsr<12> = bit IMPLEMENTATION_DEFINED "Virtual External abort type";
    if TTBCR.EAE == '1' then // Long-descriptor format
        fsr<9> = '1';
        fsr<5:0> = EncodeLDFSC(fault, level);
    else                     // Short-descriptor format
        fsr<9> = '0';
        fsr<10,3:0> = EncodeSDFSC(fault, level);
    DFSR = fsr;
    DFAR = bits(32) UNKNOWN;
    ClearPendingVirtualSError();
    AArch32.EnterMode(M32_Abort, preferred_exception_return, lr_offset, vect_offset);
```

Library pseudocode for aarch32/exceptions/debug/AArch32.SoftwareBreakpoint

```
// AArch32.SoftwareBreakpoint()
// ============================

AArch32.SoftwareBreakpoint(bits(16) immediate)

    if (EL2Enabled() && !ELUsingAArch32(EL2) &&
        (HCR_EL2.TGE == '1' || MDCR_EL2.TDE == '1')) || !ELUsingAArch32(EL1) then
        AArch64.SoftwareBreakpoint(immediate);

    accdesc  = CreateAccDescIFetch();
    fault    = NoFault(accdesc);
    vaddress = bits(32) UNKNOWN;

    fault.statuscode = Fault_Debug;
    fault.debugmoe   = DebugException_BKPT;

    AArch32.Abort(vaddress, fault);
```

Library pseudocode for aarch32/exceptions/debug/DebugException

```
constant bits(4) DebugException_Breakpoint  = '0001';
constant bits(4) DebugException_BKPT        = '0011';
constant bits(4) DebugException_VectorCatch = '0101';
constant bits(4) DebugException_Watchpoint  = '1010';
```

Library pseudocode for aarch32/exceptions/exceptions/ AArch32.CheckAdvSIMDOrFPRegisterTraps

```
// AArch32.CheckAdvSIMDOrFPRegisterTraps()
// =======================================
// Check if an instruction that accesses an Advanced SIMD and
// floating-point System register is trapped by an appropriate HCR.TIDx
// ID group trap control.

AArch32.CheckAdvSIMDOrFPRegisterTraps(bits(4) reg)

    if PSTATE.EL == EL1 && EL2Enabled() then
        tid0 = if ELUsingAArch32(EL2) then HCR.TID0 else HCR_EL2.TID0;
        tid3 = if ELUsingAArch32(EL2) then HCR.TID3 else HCR_EL2.TID3;

        if ((tid0 == '1' && reg == '0000') ||                            // FPSID
              (tid3 == '1' && reg IN {'0101', '0110', '0111'})) then     // MVFRx
            if ELUsingAArch32(EL2) then
                AArch32.SystemAccessTrap(M32_Hyp, 0x8);
            else
                AArch64.AArch32SystemAccessTrap(EL2, 0x8);
```

Library pseudocode for aarch32/exceptions/exceptions/AArch32.ExceptionClass

```
// AArch32.ExceptionClass()
// ========================
// Returns the Exception Class and Instruction Length fields to be reported in HSR

(integer,bit) AArch32.ExceptionClass(Exception exceptype)

    il_is_valid = TRUE;
    integer ec;
    case exceptype of
        when Exception_Uncategorized         ec = 0x00; il_is_valid = FALSE;
        when Exception_WFxTrap               ec = 0x01;
        when Exception_CP15RTTrap            ec = 0x03;
        when Exception_CP15RRTTrap           ec = 0x04;
        when Exception_CP14RTTrap            ec = 0x05;
        when Exception_CP14DTTrap            ec = 0x06;
        when Exception_AdvSIMDFPAccessTrap   ec = 0x07;
        when Exception_FPIDTrap              ec = 0x08;
        when Exception_PACTrap               ec = 0x09;
        when Exception_TSTARTAccessTrap      ec = 0x1B;
        when Exception_GPC                   ec = 0x1E;
        when Exception_CP14RRTTrap           ec = 0x0C;
        when Exception_BranchTarget          ec = 0x0D;
        when Exception_IllegalState          ec = 0x0E; il_is_valid = FALSE;
        when Exception_SupervisorCall        ec = 0x11;
        when Exception_HypervisorCall        ec = 0x12;
        when Exception_MonitorCall           ec = 0x13;
        when Exception_InstructionAbort      ec = 0x20; il_is_valid = FALSE;
        when Exception_PCAlignment           ec = 0x22; il_is_valid = FALSE;
        when Exception_DataAbort             ec = 0x24;
        when Exception_NV2DataAbort          ec = 0x25;
        when Exception_FPTrappedException    ec = 0x28;
        otherwise                            Unreachable();

    if ec IN {0x20,0x24} && PSTATE.EL == EL2 then
        ec = ec + 1;
    bit il;
    if il_is_valid then
        il = if ThisInstrLength() == 32 then '1' else '0';
    else
        il = '1';

    return (ec,il);
```

Library pseudocode for aarch32/exceptions/exceptions/AArch32.GeneralExceptionsToAArch64

```
// AArch32.GeneralExceptionsToAArch64()
// ====================================
// Returns TRUE if exceptions normally routed to EL1 are being handled at an Exception
// level using AArch64, because either EL1 is using AArch64 or TGE is in force and EL2
// is using AArch64.

boolean AArch32.GeneralExceptionsToAArch64()
    return ((PSTATE.EL == EL0 && !ELUsingAArch32(EL1)) ||
            (EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1'));
```

Library pseudocode for aarch32/exceptions/exceptions/AArch32.ReportHypEntry

```
// AArch32.ReportHypEntry()
// ========================
// Report syndrome information to Hyp mode registers.

AArch32.ReportHypEntry(ExceptionRecord except)

    Exception exceptype = except.exceptype;

    (ec,il) = AArch32.ExceptionClass(exceptype);
    iss  = except.syndrome;
    iss2 = except.syndrome2;

    // IL is not valid for Data Abort exceptions without valid instruction syndrome information
    if ec IN {0x24,0x25} && iss<24> == '0' then
        il = '1';

    HSR = ec<5:0>:il:iss;

    if exceptype IN {Exception_InstructionAbort, Exception_PCAlignment} then
        HIFAR = except.vaddress<31:0>;
        HDFAR = bits(32) UNKNOWN;
    elsif exceptype == Exception_DataAbort then
        HIFAR = bits(32) UNKNOWN;
        HDFAR = except.vaddress<31:0>;

    if except.ipavalid then
        HPFAR<31:4> = except.ipaddress<39:12>;
    else
        HPFAR<31:4> = bits(28) UNKNOWN;

    return;
```

Library pseudocode for aarch32/exceptions/exceptions/AArch32.ResetControlRegisters

```
// AArch32.ResetControlRegisters()
// ===============================
// Resets System registers and memory-mapped control registers that have architecturally-defined
// reset values to those values.

AArch32.ResetControlRegisters(boolean cold_reset);
```

Library pseudocode for aarch32/exceptions/exceptions/AArch32.TakeReset

```
// AArch32.TakeReset()
// ===================
// Reset into AArch32 state

AArch32.TakeReset(boolean cold_reset)
    assert !HaveAArch64();

    // Enter the highest implemented Exception level in AArch32 state
    if HaveEL(EL3) then
        AArch32.WriteMode(M32_Svc);
        SCR.NS = '0';                     // Secure state
    elsif HaveEL(EL2) then
        AArch32.WriteMode(M32_Hyp);
    else
        AArch32.WriteMode(M32_Svc);

    // Reset System registers in the coproc=0b111x encoding space
    // and other system components
    AArch32.ResetControlRegisters(cold_reset);
    FPEXC.EN = '0';

    // Reset all other PSTATE fields, including instruction set and endianness according to the
    // SCTLR values produced by the above call to ResetControlRegisters()
    PSTATE.<A,I,F> = '111';             // All asynchronous exceptions masked
    PSTATE.IT = '00000000';             // IT block state reset
    if HaveEL(EL2) && !HaveEL(EL3) then
        PSTATE.T = HSCTLR.TE;           // Instruction set: TE=0:A32, TE=1:T32. PSTATE.J is RES0.
        PSTATE.E = HSCTLR.EE;           // Endianness: EE=0: little-endian, EE=1: big-endian.
    else
        PSTATE.T = SCTLR.TE;            // Instruction set: TE=0:A32, TE=1:T32. PSTATE.J is RES0.
        PSTATE.E = SCTLR.EE;            // Endianness: EE=0: little-endian, EE=1: big-endian.
    PSTATE.IL = '0';                    // Clear Illegal Execution state bit

    // All registers, bits and fields not reset by the above pseudocode or by the BranchTo() call
    // below are UNKNOWN bitstrings after reset. In particular, the return information registers
    // R14 or ELR_hyp and SPSR have UNKNOWN values, so that it
    // is impossible to return from a reset in an architecturally defined way.
    AArch32.ResetGeneralRegisters();
    AArch32.ResetSIMDFPRegisters();
    AArch32.ResetSpecialRegisters();
    ResetExternalDebugRegisters(cold_reset);

    bits(32) rv;                      // IMPLEMENTATION DEFINED reset vector

    if HaveEL(EL3) then
        if MVBAR<0> == '1' then           // Reset vector in MVBAR
            rv = MVBAR<31:1>:'0';
        else
            rv = bits(32) IMPLEMENTATION_DEFINED "reset vector address";
    else
        rv = RVBAR<31:1>:'0';

    // The reset vector must be correctly aligned
    assert rv<0> == '0' && (PSTATE.T == '1' || rv<1> == '0');

    boolean branch_conditional = FALSE;
    BranchTo(rv, BranchType_RESET, branch_conditional);
```

Library pseudocode for aarch32/exceptions/exceptions/ExcVectorBase

```
// ExcVectorBase()
// ===============

bits(32) ExcVectorBase()
    if SCTLR.V == '1' then  // Hivecs selected, base = 0xFFFF0000
        return Ones(16):Zeros(16);
    else
        return VBAR<31:5>:Zeros(5);
```

Library pseudocode for aarch32/exceptions/ieeefp/AArch32.FPTrappedException

```
// AArch32.FPTrappedException()
// ============================

AArch32.FPTrappedException(bits(8) accumulated_exceptions)
    if AArch32.GeneralExceptionsToAArch64() then
        is_ase = FALSE;
        element = 0;
        AArch64.FPTrappedException(is_ase, accumulated_exceptions);
    FPEXC.DEX    = '1';
    FPEXC.TFV    = '1';
    FPEXC<7,4:0> = accumulated_exceptions<7,4:0>;                  // IDF,IXF,UFF,OFF,DZF,IOF
    FPEXC<10:8>  = '111';                                          // VECITR is RES1

    AArch32.TakeUndefInstrException();
```

Library pseudocode for aarch32/exceptions/syscalls/AArch32.CallHypervisor

```
// AArch32.CallHypervisor()
// ========================
// Performs a HVC call

AArch32.CallHypervisor(bits(16) immediate)
    assert HaveEL(EL2);

    if !ELUsingAArch32(EL2) then
        AArch64.CallHypervisor(immediate);
    else
        AArch32.TakeHVCException(immediate);
```

Library pseudocode for aarch32/exceptions/syscalls/AArch32.CallSupervisor

```
// AArch32.CallSupervisor()
// ========================
// Calls the Supervisor

AArch32.CallSupervisor(bits(16) immediate_in)
    bits(16) immediate = immediate_in;
    if AArch32.CurrentCond() != '1110' then
        immediate = bits(16) UNKNOWN;
    if AArch32.GeneralExceptionsToAArch64() then
        AArch64.CallSupervisor(immediate);
    else
        AArch32.TakeSVCException(immediate);
```

Library pseudocode for aarch32/exceptions/syscalls/AArch32.TakeHVCException

```
// AArch32.TakeHVCException()
// ==========================

AArch32.TakeHVCException(bits(16) immediate)
    assert HaveEL(EL2) && ELUsingAArch32(EL2);

    AArch32.ITAdvance();
    SSAdvance();
    bits(32) preferred_exception_return = NextInstrAddr(32);
    vect_offset = 0x08;

    except = ExceptionSyndrome(Exception_HypervisorCall);
    except.syndrome<15:0> = immediate;

    if PSTATE.EL == EL2 then
        AArch32.EnterHypMode(except, preferred_exception_return, vect_offset);
    else
        AArch32.EnterHypMode(except, preferred_exception_return, 0x14);
```

Library pseudocode for aarch32/exceptions/syscalls/AArch32.TakeSMCException

```
// AArch32.TakeSMCException()
// ==========================

AArch32.TakeSMCException()
    assert HaveEL(EL3) && ELUsingAArch32(EL3);
    AArch32.ITAdvance();
    SSAdvance();
    bits(32) preferred_exception_return = NextInstrAddr(32);
    vect_offset = 0x08;
    lr_offset = 0;

    AArch32.EnterMonitorMode(preferred_exception_return, lr_offset, vect_offset);
```

Library pseudocode for aarch32/exceptions/syscalls/AArch32.TakeSVCException

```
// AArch32.TakeSVCException()
// ==========================

AArch32.TakeSVCException(bits(16) immediate)

    AArch32.ITAdvance();
    SSAdvance();
    route_to_hyp = PSTATE.EL == EL0 && EL2Enabled() && HCR.TGE == '1';

    bits(32) preferred_exception_return = NextInstrAddr(32);
    vect_offset = 0x08;
    lr_offset = 0;

    if PSTATE.EL == EL2 || route_to_hyp then
        except = ExceptionSyndrome(Exception_SupervisorCall);
        except.syndrome<15:0> = immediate;
        if PSTATE.EL == EL2 then
            AArch32.EnterHypMode(except, preferred_exception_return, vect_offset);
        else
            AArch32.EnterHypMode(except, preferred_exception_return, 0x14);
    else
        AArch32.EnterMode(M32_Svc, preferred_exception_return, lr_offset, vect_offset);
```

Library pseudocode for aarch32/exceptions/takeexception/AArch32.EnterHypMode

```
// AArch32.EnterHypMode()
// ======================
// Take an exception to Hyp mode.

AArch32.EnterHypMode(ExceptionRecord except, bits(32) preferred_exception_return,
                     integer vect_offset)
    SynchronizeContext();
    assert HaveEL(EL2) && CurrentSecurityState() == SS_NonSecure && ELUsingAArch32(EL2);

    if Halted() then
        AArch32.EnterHypModeInDebugState(except);
        return;
    bits(32) spsr = GetPSRFromPSTATE(AArch32_NonDebugState, 32);
    if !(except.exceptype IN {Exception_IRQ, Exception_FIQ}) then
        AArch32.ReportHypEntry(except);
    AArch32.WriteMode(M32_Hyp);
    SPSR[] = spsr;
    ELR_hyp = preferred_exception_return;
    PSTATE.T = HSCTLR.TE;                       // PSTATE.J is RES0
    PSTATE.SS = '0';
    if !HaveEL(EL3) || SCR_GEN[].EA == '0' then PSTATE.A = '1';
    if !HaveEL(EL3) || SCR_GEN[].IRQ == '0' then PSTATE.I = '1';
    if !HaveEL(EL3) || SCR_GEN[].FIQ == '0' then PSTATE.F = '1';
    PSTATE.E = HSCTLR.EE;
    PSTATE.IL = '0';
    PSTATE.IT = '00000000';
    if HaveSSBSExt() then PSTATE.SSBS = HSCTLR.DSSBS;
    boolean branch_conditional = FALSE;
    BranchTo(HVBAR<31:5>:vect_offset<4:0>, BranchType_EXCEPTION, branch_conditional);

    CheckExceptionCatch(TRUE);                  // Check for debug event on exception entry

    EndOfInstruction();
```

Library pseudocode for aarch32/exceptions/takeexception/AArch32.EnterMode

```
// AArch32.EnterMode()
// ===================
// Take an exception to a mode other than Monitor and Hyp mode.

AArch32.EnterMode(bits(5) target_mode, bits(32) preferred_exception_return, integer lr_offset,
                  integer vect_offset)
    SynchronizeContext();
    assert ELUsingAArch32(EL1) && PSTATE.EL != EL2;

    if Halted() then
        AArch32.EnterModeInDebugState(target_mode);
        return;
    bits(32) spsr = GetPSRFromPSTATE(AArch32_NonDebugState, 32);
    if PSTATE.M == M32_Monitor then SCR.NS = '0';
    AArch32.WriteMode(target_mode);
    SPSR[] = spsr;
    R[14] = preferred_exception_return + lr_offset;
    PSTATE.T = SCTLR.TE;                        // PSTATE.J is RES0
    PSTATE.SS = '0';
    if target_mode == M32_FIQ then
        PSTATE.<A,I,F> = '111';
    elsif target_mode IN {M32_Abort, M32_IRQ} then
        PSTATE.<A,I> = '11';
    else
        PSTATE.I = '1';
    PSTATE.E = SCTLR.EE;
    PSTATE.IL = '0';
    PSTATE.IT = '00000000';
    if HavePANExt() && SCTLR.SPAN == '0' then PSTATE.PAN = '1';
    if HaveSSBSExt() then PSTATE.SSBS = SCTLR.DSSBS;
    boolean branch_conditional = FALSE;
    BranchTo(ExcVectorBase()<31:5>:vect_offset<4:0>, BranchType_EXCEPTION, branch_conditional);

    CheckExceptionCatch(TRUE);                  // Check for debug event on exception entry

    EndOfInstruction();
```

Library pseudocode for aarch32/exceptions/takeexception/AArch32.EnterMonitorMode

```
// AArch32.EnterMonitorMode()
// ==========================
// Take an exception to Monitor mode.

AArch32.EnterMonitorMode(bits(32) preferred_exception_return, integer lr_offset,
                         integer vect_offset)
    SynchronizeContext();
    assert HaveEL(EL3) && ELUsingAArch32(EL3);
    from_secure = CurrentSecurityState() == SS_Secure;
    if Halted() then
        AArch32.EnterMonitorModeInDebugState();
        return;
    bits(32) spsr = GetPSRFromPSTATE(AArch32_NonDebugState, 32);
    if PSTATE.M == M32_Monitor then SCR.NS = '0';
    AArch32.WriteMode(M32_Monitor);
    SPSR[] = spsr;
    R[14] = preferred_exception_return + lr_offset;
    PSTATE.T = SCTLR.TE;                        // PSTATE.J is RES0
    PSTATE.SS = '0';
    PSTATE.<A,I,F> = '111';
    PSTATE.E = SCTLR.EE;
    PSTATE.IL = '0';
    PSTATE.IT = '00000000';
    if HavePANExt() then
        if !from_secure then
            PSTATE.PAN = '0';
        elsif SCTLR.SPAN == '0' then
            PSTATE.PAN = '1';
    if HaveSSBSExt() then PSTATE.SSBS = SCTLR.DSSBS;
    boolean branch_conditional = FALSE;
    BranchTo(MVBAR<31:5>:vect_offset<4:0>, BranchType_EXCEPTION, branch_conditional);

    CheckExceptionCatch(TRUE);                  // Check for debug event on exception entry

    EndOfInstruction();
```

Library pseudocode for aarch32/exceptions/traps/AArch32.CheckAdvSIMDOrFPEnabled

```
// AArch32.CheckAdvSIMDOrFPEnabled()
// =================================
// Check against CPACR, FPEXC, HCPTR, NSACR, and CPTR_EL3.

AArch32.CheckAdvSIMDOrFPEnabled(boolean fpexc_check, boolean advsimd)
    if (PSTATE.EL == EL0 && !ELUsingAArch32(EL1) &&
          (!EL2Enabled() || (!ELUsingAArch32(EL2) && HCR_EL2.TGE == '0'))) then
        // The PE behaves as if FPEXC.EN is 1
        AArch64.CheckFPEnabled();
        AArch64.CheckFPAdvSIMDEnabled();
    elsif (PSTATE.EL == EL0 && EL2Enabled() &&
             !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' && !ELUsingAArch32(EL1)) then
        if fpexc_check && HCR_EL2.RW == '0' then
            fpexc_en = bits(1) IMPLEMENTATION_DEFINED "FPEXC.EN value when TGE==1 and RW==0";
            if fpexc_en == '0' then UNDEFINED;
        AArch64.CheckFPEnabled();
    else
        cpacr_asedis = CPACR.ASEDIS;
        cpacr_cp10 = CPACR.cp10;

        if HaveEL(EL3) && ELUsingAArch32(EL3) && CurrentSecurityState() == SS_NonSecure then
            // Check if access disabled in NSACR
            if NSACR.NSASEDIS == '1' then cpacr_asedis = '1';
            if NSACR.cp10 == '0' then cpacr_cp10 = '00';

        if PSTATE.EL != EL2 then
            // Check if Advanced SIMD disabled in CPACR
            if advsimd && cpacr_asedis == '1' then UNDEFINED;

            // Check if access disabled in CPACR
            boolean disabled;
            case cpacr_cp10 of
                when '00' disabled = TRUE;
                when '01' disabled = PSTATE.EL == EL0;
                when '10' disabled = ConstrainUnpredictableBool(Unpredictable_RESCPACR);
                when '11' disabled = FALSE;
            if disabled then UNDEFINED;

        // If required, check FPEXC enabled bit.
        if fpexc_check && FPEXC.EN == '0' then UNDEFINED;

        AArch32.CheckFPAdvSIMDTrap(advsimd);    // Also check against HCPTR and CPTR_EL3
```

Library pseudocode for aarch32/exceptions/traps/AArch32.CheckFPAdvSIMDTrap

```
// AArch32.CheckFPAdvSIMDTrap()
// ============================
// Check against CPTR_EL2 and CPTR_EL3.

AArch32.CheckFPAdvSIMDTrap(boolean advsimd)
    if EL2Enabled() && !ELUsingAArch32(EL2) then
        AArch64.CheckFPAdvSIMDTrap();
    else
        if (HaveEL(EL3) && !ELUsingAArch32(EL3) &&
            CPTR_EL3.TFP == '1' && EL3SDDUndefPriority()) then
            UNDEFINED;

        ss = CurrentSecurityState();
        if HaveEL(EL2) && ss != SS_Secure then
            hcptr_tase = HCPTR.TASE;
            hcptr_cp10 = HCPTR.TCP10;

            if HaveEL(EL3) && ELUsingAArch32(EL3) then
                // Check if access disabled in NSACR
                if NSACR.NSASEDIS == '1' then hcptr_tase = '1';
                if NSACR.cp10 == '0' then hcptr_cp10 = '1';

            // Check if access disabled in HCPTR
            if (advsimd && hcptr_tase == '1') || hcptr_cp10 == '1' then
                except = ExceptionSyndrome(Exception_AdvSIMDFPAccessTrap);
                except.syndrome<24:20> = ConditionSyndrome();

                if advsimd then
                    except.syndrome<5> = '1';
                else
                    except.syndrome<5> = '0';
                    except.syndrome<3:0> = '1010';         // coproc field, always 0xA

                if PSTATE.EL == EL2 then
                    AArch32.TakeUndefInstrException(except);
                else
                    AArch32.TakeHypTrapException(except);

        if HaveEL(EL3) && !ELUsingAArch32(EL3) then
            // Check if access disabled in CPTR_EL3
            if CPTR_EL3.TFP == '1' then
                if EL3SDDUndef() then
                    UNDEFINED;
                else
                    AArch64.AdvSIMDFPAccessTrap(EL3);
```

Library pseudocode for aarch32/exceptions/traps/AArch32.CheckForSMCUndefOrTrap

```
// AArch32.CheckForSMCUndefOrTrap()
// ================================
// Check for UNDEFINED or trap on SMC instruction

AArch32.CheckForSMCUndefOrTrap()
    if !HaveEL(EL3) || PSTATE.EL == EL0 then
        UNDEFINED;

    if EL2Enabled() && !ELUsingAArch32(EL2) then
        AArch64.CheckForSMCUndefOrTrap(Zeros(16));
    else
        route_to_hyp = EL2Enabled() && PSTATE.EL == EL1 && HCR.TSC == '1';
        if route_to_hyp then
            except = ExceptionSyndrome(Exception_MonitorCall);
            AArch32.TakeHypTrapException(except);
```

Library pseudocode for aarch32/exceptions/traps/AArch32.CheckForSVCTrap

```
// AArch32.CheckForSVCTrap()
// =========================
// Check for trap on SVC instruction

AArch32.CheckForSVCTrap(bits(16) immediate)
    if HaveFGTExt() then
        route_to_el2 = FALSE;
        if PSTATE.EL == EL0 then
            route_to_el2 = (!ELUsingAArch32(EL1) && EL2Enabled() && HFGITR_EL2.SVC_EL0 == '1' &&
                           (HCR_EL2.<E2H, TGE> != '11' && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1')));

        if route_to_el2 then
            except = ExceptionSyndrome(Exception_SupervisorCall);
            except.syndrome<15:0> = immediate;
            except.trappedsyscallinst = TRUE;
            bits(64) preferred_exception_return = ThisInstrAddr(64);
            vect_offset = 0x0;

            AArch64.TakeException(EL2, except, preferred_exception_return, vect_offset);
```

Library pseudocode for aarch32/exceptions/traps/AArch32.CheckForWFxTrap

```
// AArch32.CheckForWFxTrap()
// =========================
// Check for trap on WFE or WFI instruction

AArch32.CheckForWFxTrap(bits(2) target_el, WFxType wfxtype)
    assert HaveEL(target_el);

    // Check for routing to AArch64
    if !ELUsingAArch32(target_el) then
        AArch64.CheckForWFxTrap(target_el, wfxtype);
        return;

    boolean is_wfe = wfxtype == WFxType_WFE;
    boolean trap;
    case target_el of
        when EL1
            trap = (if is_wfe then SCTLR.nTWE else SCTLR.nTWI) == '0';
        when EL2
            trap = (if is_wfe then HCR.TWE else HCR.TWI) == '1';
        when EL3
            trap = (if is_wfe then SCR.TWE else SCR.TWI) == '1';

    if trap then
        if target_el == EL1 && EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.WFxTrap(wfxtype, target_el);

        if target_el == EL3 then
            AArch32.TakeMonitorTrapException();
        elsif target_el == EL2 then
            except = ExceptionSyndrome(Exception_WFxTrap);
            except.syndrome<24:20> = ConditionSyndrome();

            case wfxtype of
                when WFxType_WFI
                    except.syndrome<0> = '0';
                when WFxType_WFE
                    except.syndrome<0> = '1';

            AArch32.TakeHypTrapException(except);
        else
            AArch32.TakeUndefInstrException();
```

Library pseudocode for aarch32/exceptions/traps/AArch32.CheckITEnabled

```
// AArch32.CheckITEnabled()
// ========================
// Check whether the T32 IT instruction is disabled.

AArch32.CheckITEnabled(bits(4) mask)
    bit it_disabled;
    if PSTATE.EL == EL2 then
        it_disabled = HSCTLR.ITD;
    else
        it_disabled = (if ELUsingAArch32(EL1) then SCTLR.ITD else SCTLR[].ITD);
    if it_disabled == '1' then
        if mask != '1000' then UNDEFINED;

        accdesc = CreateAccDescIFetch();
        aligned = TRUE;
        // Otherwise whether the IT block is allowed depends on hw1 of the next instruction.
        next_instr = AArch32.MemSingle[NextInstrAddr(32), 2, accdesc, aligned];

        if next_instr IN {'11xxxxxxxxxxxxxx', '1011xxxxxxxxxxxx', '10100xxxxxxxxxxx',
                          '01001xxxxxxxxxxx', '010001xxx1111xxx', '010001xx1xxxx111'} then
            // It is IMPLEMENTATION DEFINED whether the Undefined Instruction exception is
            // taken on the IT instruction or the next instruction. This is not reflected in
            // the pseudocode, which always takes the exception on the IT instruction. This
            // also does not take into account cases where the next instruction is UNPREDICTABLE.
            UNDEFINED;

    return;
```

Library pseudocode for aarch32/exceptions/traps/AArch32.CheckIllegalState

```
// AArch32.CheckIllegalState()
// ===========================
// Check PSTATE.IL bit and generate Illegal Execution state exception if set.

AArch32.CheckIllegalState()
    if AArch32.GeneralExceptionsToAArch64() then
        AArch64.CheckIllegalState();
    elsif PSTATE.IL == '1' then
        route_to_hyp = PSTATE.EL == EL0 && EL2Enabled() && HCR.TGE == '1';

        bits(32) preferred_exception_return = ThisInstrAddr(32);
        vect_offset = 0x04;

        if PSTATE.EL == EL2 || route_to_hyp then
            except = ExceptionSyndrome(Exception_IllegalState);
            if PSTATE.EL == EL2 then
                AArch32.EnterHypMode(except, preferred_exception_return, vect_offset);
            else
                AArch32.EnterHypMode(except, preferred_exception_return, 0x14);
        else
            AArch32.TakeUndefInstrException();
```

Library pseudocode for aarch32/exceptions/traps/AArch32.CheckSETENDEnabled

```
// AArch32.CheckSETENDEnabled()
// ============================
// Check whether the AArch32 SETEND instruction is disabled.

AArch32.CheckSETENDEnabled()
    bit setend_disabled;
    if PSTATE.EL == EL2 then
        setend_disabled = HSCTLR.SED;
    else
        setend_disabled = (if ELUsingAArch32(EL1) then SCTLR.SED else SCTLR[].SED);
    if setend_disabled == '1' then
        UNDEFINED;

    return;
```

Library pseudocode for aarch32/exceptions/traps/AArch32.SystemAccessTrap

```
// AArch32.SystemAccessTrap()
// ==========================
// Trapped  System register access.

AArch32.SystemAccessTrap(bits(5) mode, integer ec)
    (valid, target_el) = ELFromM32(mode);
    assert valid && HaveEL(target_el) && target_el != EL0 && UInt(target_el) >= UInt(PSTATE.EL);

    if target_el == EL2 then
        except = AArch32.SystemAccessTrapSyndrome(ThisInstr(), ec);
        AArch32.TakeHypTrapException(except);
    else
        AArch32.TakeUndefInstrException();
```

Library pseudocode for aarch32/exceptions/traps/AArch32.SystemAccessTrapSyndrome

```
// AArch32.SystemAccessTrapSyndrome()
// ==================================
// Returns the syndrome information for traps on AArch32 MCR, MCRR, MRC, MRRC, and VMRS,
// VMSR instructions, other than traps that are due to HCPTR or CPACR.

ExceptionRecord AArch32.SystemAccessTrapSyndrome(bits(32) instr, integer ec)
    ExceptionRecord except;

    case ec of
        when 0x0    except = ExceptionSyndrome(Exception_Uncategorized);
        when 0x3    except = ExceptionSyndrome(Exception_CP15RTTrap);
        when 0x4    except = ExceptionSyndrome(Exception_CP15RRTTrap);
        when 0x5    except = ExceptionSyndrome(Exception_CP14RTTrap);
        when 0x6    except = ExceptionSyndrome(Exception_CP14DTTrap);
        when 0x7    except = ExceptionSyndrome(Exception_AdvSIMDFPAccessTrap);
        when 0x8    except = ExceptionSyndrome(Exception_FPIDTrap);
        when 0xC    except = ExceptionSyndrome(Exception_CP14RRTTrap);
        otherwise   Unreachable();

    bits(20) iss = Zeros(20);

    if except.exceptype == Exception_Uncategorized then
        return except;
    elsif except.exceptype IN {Exception_FPIDTrap, Exception_CP14RTTrap,
                                  Exception_CP15RTTrap} then
        // Trapped MRC/MCR, VMRS on FPSID
        iss<13:10> = instr<19:16>;        // CRn, Reg in case of VMRS
        iss<8:5>   = instr<15:12>;        // Rt
        iss<9>     = '0';                 // RES0

        if except.exceptype != Exception_FPIDTrap then    // When trap is not for VMRS
            iss<19:17> = instr<7:5>;          // opc2
            iss<16:14> = instr<23:21>;        // opc1
            iss<4:1>   = instr<3:0>;          //CRm
        else //VMRS Access
            iss<19:17> = '000';               //opc2 - Hardcoded for VMRS
            iss<16:14> = '111';               //opc1 - Hardcoded for VMRS
            iss<4:1>   = '0000';              //CRm  - Hardcoded for VMRS
    elsif except.exceptype IN {Exception_CP14RRTTrap, Exception_AdvSIMDFPAccessTrap,
                               Exception_CP15RRTTrap} then
        // Trapped MRRC/MCRR, VMRS/VMSR
        iss<19:16> = instr<7:4>;          // opc1
        iss<13:10> = instr<19:16>;        // Rt2
        iss<8:5> = instr<15:12>;          // Rt
        iss<4:1>   = instr<3:0>;         // CRm
    elsif except.exceptype == Exception_CP14DTTrap then
        // Trapped LDC/STC
        iss<19:12> = instr<7:0>;         // imm8
        iss<4>     = instr<23>;          // U
        iss<2:1>   = instr<24,21>;       // P,W
        if instr<19:16> == '1111' then   // Rn==15, LDC(Literal addressing)/STC
            iss<8:5> = bits(4) UNKNOWN;
            iss<3>   = '1';
    iss<0> = instr<20>;                  // Direction

    except.syndrome<24:20> = ConditionSyndrome();
    except.syndrome<19:0>  = iss;

    return except;
```

Library pseudocode for aarch32/exceptions/traps/AArch32.TakeHypTrapException

```
// AArch32.TakeHypTrapException()
// ==============================
// Exceptions routed to Hyp mode as a Hyp Trap exception.

AArch32.TakeHypTrapException(integer ec)
    except = AArch32.SystemAccessTrapSyndrome(ThisInstr(), ec);
    AArch32.TakeHypTrapException(except);

// AArch32.TakeHypTrapException()
// ==============================
// Exceptions routed to Hyp mode as a Hyp Trap exception.

AArch32.TakeHypTrapException(ExceptionRecord except)
    assert HaveEL(EL2) && CurrentSecurityState() == SS_NonSecure && ELUsingAArch32(EL2);

    bits(32) preferred_exception_return = ThisInstrAddr(32);
    vect_offset = 0x14;

    AArch32.EnterHypMode(except, preferred_exception_return, vect_offset);
```

Library pseudocode for aarch32/exceptions/traps/AArch32.TakeMonitorTrapException

```
// AArch32.TakeMonitorTrapException()
// ==================================
// Exceptions routed to Monitor mode as a Monitor Trap exception.

AArch32.TakeMonitorTrapException()
    assert HaveEL(EL3) && ELUsingAArch32(EL3);

    bits(32) preferred_exception_return = ThisInstrAddr(32);
    vect_offset = 0x04;
    lr_offset = if CurrentInstrSet() == InstrSet_A32 then 4 else 2;

    AArch32.EnterMonitorMode(preferred_exception_return, lr_offset, vect_offset);
```

Library pseudocode for aarch32/exceptions/traps/AArch32.TakeUndefInstrException

```
// AArch32.TakeUndefInstrException()
// =================================

AArch32.TakeUndefInstrException()
    except = ExceptionSyndrome(Exception_Uncategorized);
    AArch32.TakeUndefInstrException(except);

// AArch32.TakeUndefInstrException()
// =================================

AArch32.TakeUndefInstrException(ExceptionRecord except)

    route_to_hyp = PSTATE.EL == EL0 && EL2Enabled() && HCR.TGE == '1';
    bits(32) preferred_exception_return = ThisInstrAddr(32);
    vect_offset = 0x04;
    lr_offset = if CurrentInstrSet() == InstrSet_A32 then 4 else 2;

    if PSTATE.EL == EL2 then
        AArch32.EnterHypMode(except, preferred_exception_return, vect_offset);
    elsif route_to_hyp then
        AArch32.EnterHypMode(except, preferred_exception_return, 0x14);
    else
        AArch32.EnterMode(M32_Undef, preferred_exception_return, lr_offset, vect_offset);
```

Library pseudocode for aarch32/exceptions/traps/AArch32.UndefinedFault

```
// AArch32.UndefinedFault()
// ========================

AArch32.UndefinedFault()

    if AArch32.GeneralExceptionsToAArch64() then AArch64.UndefinedFault();
    AArch32.TakeUndefInstrException();
```

Library pseudocode for aarch32/functions/aborts/AArch32.DomainValid

```
// AArch32.DomainValid()
// =====================
// Returns TRUE if the Domain is valid for a Short-descriptor translation scheme.

boolean AArch32.DomainValid(Fault statuscode, integer level)
    assert statuscode != Fault_None;

    case statuscode of
        when Fault_Domain
            return TRUE;
        when Fault_Translation, Fault_AccessFlag, Fault_SyncExternalOnWalk, Fault_SyncParityOnWalk
            return level == 2;
        otherwise
            return FALSE;
```

Library pseudocode for aarch32/functions/aborts/AArch32.FaultSyndrome

```
// AArch32.FaultSyndrome()
// =======================
// Creates an exception syndrome value for Abort and Watchpoint exceptions taken to
// AArch32 Hyp mode.

bits(25) AArch32.FaultSyndrome(boolean d_side, FaultRecord fault)
    assert fault.statuscode != Fault_None;

    bits(25) iss  = Zeros(25);
    bits(24) iss2 = Zeros(24);

    if HaveRASExt() && IsAsyncAbort(fault) then
        ErrorState errstate = AArch32.PEErrorState(fault);
        iss<11:10> = AArch32.EncodeAsyncErrorSyndrome(errstate); // AET

    if d_side then
        if (IsSecondStage(fault) && !fault.s2fs1walk &&
            (!IsExternalSyncAbort(fault) ||
            (!HaveRASExt() && fault.access.acctype == AccessType_TTW &&
            boolean IMPLEMENTATION_DEFINED "ISV on second stage translation table walk"))) then
            iss<24:14> = LSInstructionSyndrome();

        if fault.access.acctype IN {AccessType_DC, AccessType_IC, AccessType_AT} then
            iss<8> = '1';

        if fault.access.acctype IN {AccessType_DC, AccessType_IC, AccessType_AT} then
            iss<6> = '1';
        elsif fault.statuscode IN {Fault_HWUpdateAccessFlag, Fault_Exclusive} then
            iss<6> = bit UNKNOWN;
        elsif fault.access.atomicop && IsExternalAbort(fault) then
            iss<6> = bit UNKNOWN;
        else
            iss<6> = if fault.write then '1' else '0';

    if IsExternalAbort(fault) then iss<9> = fault.extflag;
    iss<7> = if fault.s2fs1walk then '1' else '0';
    iss<5:0> = EncodeLDFSC(fault.statuscode, fault.level);

    return (iss);
```

Library pseudocode for aarch32/functions/aborts/EncodeSDFSC

```
// EncodeSDFSC()
// =============
// Function that gives the Short-descriptor FSR code for different types of Fault

bits(5) EncodeSDFSC(Fault statuscode, integer level)

    bits(5) result;
    case statuscode of
        when Fault_AccessFlag
            assert level IN {1,2};
            result = if level == 1 then '00011' else '00110';
        when Fault_Alignment
            result = '00001';
        when Fault_Permission
            assert level IN {1,2};
            result = if level == 1 then '01101' else '01111';
        when Fault_Domain
            assert level IN {1,2};
            result = if level == 1 then '01001' else '01011';
        when Fault_Translation
            assert level IN {1,2};
            result = if level == 1 then '00101' else '00111';
        when Fault_SyncExternal
            result = '01000';
        when Fault_SyncExternalOnWalk
            assert level IN {1,2};
            result = if level == 1 then '01100' else '01110';
        when Fault_SyncParity
            result = '11001';
        when Fault_SyncParityOnWalk
            assert level IN {1,2};
            result = if level == 1 then '11100' else '11110';
        when Fault_AsyncParity
            result = '11000';
        when Fault_AsyncExternal
            result = '10110';
        when Fault_Debug
            result = '00010';
        when Fault_TLBConflict
            result = '10000';
        when Fault_Lockdown
            result = '10100';   // IMPLEMENTATION DEFINED
        when Fault_Exclusive
            result = '10101';   // IMPLEMENTATION DEFINED
        when Fault_ICacheMaint
            result = '00100';
        otherwise
            Unreachable();

    return result;
```

Library pseudocode for aarch32/functions/common/A32ExpandImm

```
// A32ExpandImm()
// ==============

bits(32) A32ExpandImm(bits(12) imm12)

    // PSTATE.C argument to following function call does not affect the imm32 result.
    (imm32, -) = A32ExpandImm_C(imm12, PSTATE.C);

    return imm32;
```

Library pseudocode for aarch32/functions/common/A32ExpandImm_C

```
// A32ExpandImm_C()
// ================

(bits(32), bit) A32ExpandImm_C(bits(12) imm12, bit carry_in)

    unrotated_value = ZeroExtend(imm12<7:0>, 32);
    (imm32, carry_out) = Shift_C(unrotated_value, SRType_ROR, 2*UInt(imm12<11:8>), carry_in);

    return (imm32, carry_out);
```

Library pseudocode for aarch32/functions/common/DecodeImmShift

```
// DecodeImmShift()
// ================

(SRType, integer) DecodeImmShift(bits(2) srtype, bits(5) imm5)

    SRType shift_t;
    integer shift_n;
    case srtype of
        when '00'
            shift_t = SRType_LSL;  shift_n = UInt(imm5);
        when '01'
            shift_t = SRType_LSR;  shift_n = if imm5 == '00000' then 32 else UInt(imm5);
        when '10'
            shift_t = SRType_ASR;  shift_n = if imm5 == '00000' then 32 else UInt(imm5);
        when '11'
            if imm5 == '00000' then
                shift_t = SRType_RRX;  shift_n = 1;
            else
                shift_t = SRType_ROR;  shift_n = UInt(imm5);

    return (shift_t, shift_n);
```

Library pseudocode for aarch32/functions/common/DecodeRegShift

```
// DecodeRegShift()
// ================

SRType DecodeRegShift(bits(2) srtype)
    SRType shift_t;
    case srtype of
        when '00'  shift_t = SRType_LSL;
        when '01'  shift_t = SRType_LSR;
        when '10'  shift_t = SRType_ASR;
        when '11'  shift_t = SRType_ROR;
    return shift_t;
```

Library pseudocode for aarch32/functions/common/RRX

```
// RRX()
// =====

bits(N) RRX(bits(N) x, bit carry_in)
    (result, -) = RRX_C(x, carry_in);
    return result;
```

Library pseudocode for aarch32/functions/common/RRX_C

```
// RRX_C()
// =======

(bits(N), bit) RRX_C(bits(N) x, bit carry_in)
    result = carry_in : x<N-1:1>;
    carry_out = x<0>;
    return (result, carry_out);
```

Library pseudocode for aarch32/functions/common/SRType

```
// SRType
// ======

enumeration SRType {SRType_LSL, SRType_LSR, SRType_ASR, SRType_ROR, SRType_RRX};
```

Library pseudocode for aarch32/functions/common/Shift

```
// Shift()
// =======

bits(N) Shift(bits(N) value, SRType srtype, integer amount, bit carry_in)
    (result, -) = Shift_C(value, srtype, amount, carry_in);
    return result;
```

Library pseudocode for aarch32/functions/common/Shift_C

```
// Shift_C()
// =========

(bits(N), bit) Shift_C(bits(N) value, SRType srtype, integer amount, bit carry_in)
    assert !(srtype == SRType_RRX && amount != 1);

    bits(N) result;
    bit carry_out;
    if amount == 0 then
        (result, carry_out) = (value, carry_in);
    else
        case srtype of
            when SRType_LSL
                (result, carry_out) = LSL_C(value, amount);
            when SRType_LSR
                (result, carry_out) = LSR_C(value, amount);
            when SRType_ASR
                (result, carry_out) = ASR_C(value, amount);
            when SRType_ROR
                (result, carry_out) = ROR_C(value, amount);
            when SRType_RRX
                (result, carry_out) = RRX_C(value, carry_in);

    return (result, carry_out);
```

Library pseudocode for aarch32/functions/common/T32ExpandImm

```
// T32ExpandImm()
// ==============

bits(32) T32ExpandImm(bits(12) imm12)

    // PSTATE.C argument to following function call does not affect the imm32 result.
    (imm32, -) = T32ExpandImm_C(imm12, PSTATE.C);

    return imm32;
```

Library pseudocode for aarch32/functions/common/T32ExpandImm_C

```
// T32ExpandImm_C()
// ================

(bits(32), bit) T32ExpandImm_C(bits(12) imm12, bit carry_in)
    bits(32) imm32;
    bit carry_out;
    if imm12<11:10> == '00' then
        case imm12<9:8> of
            when '00'
                imm32 = ZeroExtend(imm12<7:0>, 32);
            when '01'
                imm32 = '00000000' : imm12<7:0> : '00000000' : imm12<7:0>;
            when '10'
                imm32 = imm12<7:0> : '00000000' : imm12<7:0> : '00000000';
            when '11'
                imm32 = imm12<7:0> : imm12<7:0> : imm12<7:0> : imm12<7:0>;
        carry_out = carry_in;
    else
        unrotated_value = ZeroExtend('1':imm12<6:0>, 32);
        (imm32, carry_out) = ROR_C(unrotated_value, UInt(imm12<11:7>));

    return (imm32, carry_out);
```

Library pseudocode for aarch32/functions/common/VBitOps

```
// VBitOps
// =======

enumeration VBitOps {VBitOps_VBIF, VBitOps_VBIT, VBitOps_VBSL};
```

Library pseudocode for aarch32/functions/common/VCGEType

```
// VCGEType
// ========

enumeration VCGEType {VCGEType_signed, VCGEType_unsigned, VCGEType_fp};
```

Library pseudocode for aarch32/functions/common/VCGTtype

```
// VCGTtype
// ========

enumeration VCGTtype {VCGTtype_signed, VCGTtype_unsigned, VCGTtype_fp};
```

Library pseudocode for aarch32/functions/common/VFPNegMul

```
// VFPNegMul
// =========

enumeration VFPNegMul {VFPNegMul_VNMLA, VFPNegMul_VNMLS, VFPNegMul_VNMUL};
```

Library pseudocode for aarch32/functions/coproc/AArch32.CheckCP15InstrCoarseTraps

```
// AArch32.CheckCP15InstrCoarseTraps()
// ===================================
// Check for coarse-grained  traps to System registers in the
// coproc=0b1111 encoding space by HSTR and HCR.

AArch32.CheckCP15InstrCoarseTraps(integer CRn, integer nreg, integer CRm)
    if PSTATE.EL == EL0 && (!ELUsingAArch32(EL1) ||
            (EL2Enabled() && !ELUsingAArch32(EL2))) then
        AArch64.CheckCP15InstrCoarseTraps(CRn, nreg, CRm);

    trapped_encoding = ((CRn == 9  && CRm IN {0,1,2,    5,6,7,8   }) ||
                        (CRn == 10 && CRm IN {0,1,    4,      8   }) ||
                        (CRn == 11 && CRm IN {0,1,2,3,4,5,6,7,8,15}));

    // Check for coarse-grained Hyp traps
    if PSTATE.EL IN {EL0, EL1} && EL2Enabled() then
        major = if nreg == 1 then CRn else CRm;
        // Check for MCR, MRC, MCRR, and MRRC disabled by HSTR<CRn/CRm>
        // and MRC and MCR disabled by HCR.TIDCP.
        if ((!(major IN {4,14}) && HSTR<major> == '1') ||
                (HCR.TIDCP == '1' && nreg == 1 && trapped_encoding)) then
            if (PSTATE.EL == EL0 &&
                    boolean IMPLEMENTATION_DEFINED "UNDEF unallocated CP15 access at EL0") then
                UNDEFINED;
            if ELUsingAArch32(EL2) then
                AArch32.SystemAccessTrap(M32_Hyp, 0x3);
            else
                AArch64.AArch32SystemAccessTrap(EL2, 0x3);
```

Library pseudocode for aarch32/functions/exclusive/AArch32.ExclusiveMonitorsPass

```
// AArch32.ExclusiveMonitorsPass()
// ===============================
// Return TRUE if the Exclusives monitors for the current PE include all of the addresses
// associated with the virtual address region of size bytes starting at address.
// The immediately following memory write must be to the same addresses.

boolean AArch32.ExclusiveMonitorsPass(bits(32) address, integer size)

    // It is IMPLEMENTATION DEFINED whether the detection of memory aborts happens
    // before or after the check on the local Exclusives monitor. As a result a failure
    // of the local monitor can occur on some implementations even if the memory
    // access would give an memory abort.
    boolean acqrel = FALSE;
    boolean tagchecked = FALSE;
    AccessDescriptor accdesc = CreateAccDescExLDST(MemOp_STORE, acqrel, tagchecked);
    boolean aligned = IsAligned(address, size);

    if !aligned then
        AArch32.Abort(address, AlignmentFault(accdesc));

    if !AArch32.IsExclusiveVA(address, ProcessorID(), size) then
        return FALSE;

    memaddrdesc = AArch32.TranslateAddress(address, accdesc, aligned, size);

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        AArch32.Abort(address, memaddrdesc.fault);

    passed = IsExclusiveLocal(memaddrdesc.paddress, ProcessorID(), size);
    ClearExclusiveLocal(ProcessorID());

    if passed && memaddrdesc.memattrs.shareability != Shareability_NSH then
        passed = IsExclusiveGlobal(memaddrdesc.paddress, ProcessorID(), size);

    return passed;
```

Library pseudocode for aarch32/functions/exclusive/AArch32.IsExclusiveVA

```
// AArch32.IsExclusiveVA()
// =======================
// An optional IMPLEMENTATION DEFINED test for an exclusive access to a virtual
// address region of size bytes starting at address.
//
// It is permitted (but not required) for this function to return FALSE and
// cause a store exclusive to fail if the virtual address region is not
// totally included within the region recorded by MarkExclusiveVA().
//
// It is always safe to return TRUE which will check the physical address only.

boolean AArch32.IsExclusiveVA(bits(32) address, integer processorid, integer size);
```

Library pseudocode for aarch32/functions/exclusive/AArch32.MarkExclusiveVA

```
// AArch32.MarkExclusiveVA()
// =========================
// Optionally record an exclusive access to the virtual address region of size bytes
// starting at address for processorid.

AArch32.MarkExclusiveVA(bits(32) address, integer processorid, integer size);
```

Library pseudocode for aarch32/functions/exclusive/AArch32.SetExclusiveMonitors

```
// AArch32.SetExclusiveMonitors()
// ==============================
// Sets the Exclusives monitors for the current PE to record the addresses associated
// with the virtual address region of size bytes starting at address.

AArch32.SetExclusiveMonitors(bits(32) address, integer size)
    boolean acqrel = FALSE;
    boolean tagchecked = FALSE;
    AccessDescriptor accdesc = CreateAccDescExLDST(MemOp_LOAD, acqrel, tagchecked);
    boolean aligned = IsAligned(address, size);

    if !aligned then
        AArch32.Abort(address, AlignmentFault(accdesc));

    memaddrdesc = AArch32.TranslateAddress(address, accdesc, aligned, size);

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        return;

    if memaddrdesc.memattrs.shareability != Shareability_NSH then
        MarkExclusiveGlobal(memaddrdesc.paddress, ProcessorID(), size);

    MarkExclusiveLocal(memaddrdesc.paddress, ProcessorID(), size);

    AArch32.MarkExclusiveVA(address, ProcessorID(), size);
```

Library pseudocode for aarch32/functions/float/CheckAdvSIMDEnabled

```
// CheckAdvSIMDEnabled()
// =====================

CheckAdvSIMDEnabled()

    fpexc_check = TRUE;
    advsimd = TRUE;

    AArch32.CheckAdvSIMDOrFPEnabled(fpexc_check, advsimd);
    // Return from CheckAdvSIMDOrFPEnabled() occurs only if Advanced SIMD access is permitted

    // Make temporary copy of D registers
    // _Dclone[] is used as input data for instruction pseudocode
    for i = 0 to 31
        _Dclone[i] = D[i];

    return;
```

Library pseudocode for aarch32/functions/float/CheckAdvSIMDOrVFPEnabled

```
// CheckAdvSIMDOrVFPEnabled()
// ==========================

CheckAdvSIMDOrVFPEnabled(boolean include_fpexc_check, boolean advsimd)
    AArch32.CheckAdvSIMDOrFPEnabled(include_fpexc_check, advsimd);
    // Return from CheckAdvSIMDOrFPEnabled() occurs only if VFP access is permitted
    return;
```

Library pseudocode for aarch32/functions/float/CheckCryptoEnabled32

```
// CheckCryptoEnabled32()
// ======================

CheckCryptoEnabled32()
    CheckAdvSIMDEnabled();
    // Return from CheckAdvSIMDEnabled() occurs only if access is permitted
    return;
```

Library pseudocode for aarch32/functions/float/CheckVFPEnabled

```
// CheckVFPEnabled()
// =================

CheckVFPEnabled(boolean include_fpexc_check)
    advsimd = FALSE;
    AArch32.CheckAdvSIMDOrFPEnabled(include_fpexc_check, advsimd);
    // Return from CheckAdvSIMDOrFPEnabled() occurs only if VFP access is permitted
    return;
```

Library pseudocode for aarch32/functions/float/FPHalvedSub

```
// FPHalvedSub()
// =============

bits(N) FPHalvedSub(bits(N) op1, bits(N) op2, FPCRType fpcr)
    assert N IN {16,32,64};
    rounding = FPRoundingMode(fpcr);
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);
    if !done then
        inf1 = (type1 == FPType_Infinity);  inf2 = (type2 == FPType_Infinity);
        zero1 = (type1 == FPType_Zero);     zero2 = (type2 == FPType_Zero);
        if inf1 && inf2 && sign1 == sign2 then
            result = FPDefaultNaN(fpcr, N);
            FPProcessException(FPExc_InvalidOp, fpcr);
        elsif (inf1 && sign1 == '0') || (inf2 && sign2 == '1') then
            result = FPInfinity('0', N);
        elsif (inf1 && sign1 == '1') || (inf2 && sign2 == '0') then
            result = FPInfinity('1', N);
        elsif zero1 && zero2 && sign1 != sign2 then
            result = FPZero(sign1, N);
        else
            result_value = (value1 - value2) / 2.0;
            if result_value == 0.0 then  // Sign of exact zero result depends on rounding mode
                result_sign = if rounding == FPRounding_NEGINF then '1' else '0';
                result = FPZero(result_sign, N);
            else
                result = FPRound(result_value, fpcr, N);
    return result;
```

Library pseudocode for aarch32/functions/float/FPRSqrtStep

```
// FPRSqrtStep()
// =============

bits(N) FPRSqrtStep(bits(N) op1, bits(N) op2)
    assert N IN {16,32};
    FPCRType fpcr = StandardFPSCRValue();
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);
    if !done then
        inf1 = (type1 == FPType_Infinity);  inf2 = (type2 == FPType_Infinity);
        zero1 = (type1 == FPType_Zero);     zero2 = (type2 == FPType_Zero);
        bits(N) product;
        if (inf1 && zero2) || (zero1 && inf2) then
            product = FPZero('0', N);
        else
            product = FPMul(op1, op2, fpcr);
        bits(N) three = FPThree('0', N);
        result = FPHalvedSub(three, product, fpcr);
    return result;
```

Library pseudocode for aarch32/functions/float/FPRecipStep

```
// FPRecipStep()
// =============

bits(N) FPRecipStep(bits(N) op1, bits(N) op2)
    assert N IN {16,32};
    FPCRType fpcr = StandardFPSCRValue();
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);
    if !done then
        inf1 = (type1 == FPType_Infinity);  inf2 = (type2 == FPType_Infinity);
        zero1 = (type1 == FPType_Zero);     zero2 = (type2 == FPType_Zero);
        bits(N) product;
        if (inf1 && zero2) || (zero1 && inf2) then
            product = FPZero('0', N);
        else
            product = FPMul(op1, op2, fpcr);
        bits(N) two = FPTwo('0', N);
        result = FPSub(two, product, fpcr);
    return result;
```

Library pseudocode for aarch32/functions/float/StandardFPSCRValue

```
// StandardFPSCRValue()
// ====================

FPCRType StandardFPSCRValue()
    bits(32) value = '00000' : FPSCR.AHP : '110000' : FPSCR.FZ16 : '0000000000000000000';
    return ZeroExtend(value, 64);
```

Library pseudocode for aarch32/functions/memory/AArch32.MemSingle

```
// AArch32.MemSingle[] - non-assignment (read) form
// ================================================
// Perform an atomic, little-endian read of 'size' bytes.

bits(size*8) AArch32.MemSingle[bits(32) address, integer size,
                               AccessDescriptor accdesc, boolean aligned]
    boolean ispair = FALSE;
    return AArch32.MemSingle[address, size, accdesc, aligned, ispair];

// AArch32.MemSingle[] - non-assignment (read) form
// ================================================
// Perform an atomic, little-endian read of 'size' bytes.

bits(size*8) AArch32.MemSingle[bits(32) address, integer size, AccessDescriptor accdesc_in,
                               boolean aligned, boolean ispair]
    assert size IN {1, 2, 4, 8, 16};
    bits(size*8) value;
    AccessDescriptor accdesc = accdesc_in;
    assert IsAligned(address, size);

    AddressDescriptor memaddrdesc;
    memaddrdesc = AArch32.TranslateAddress(address, accdesc, aligned, size);

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        AArch32.Abort(address, memaddrdesc.fault);

    // Memory array access
    if SPESampleInFlight then
        boolean is_load = TRUE;
        SPESampleLoadStore(is_load, accdesc, memaddrdesc);

    PhysMemRetStatus memstatus;
    (memstatus, value) = PhysMemRead(memaddrdesc, size, accdesc);
    if IsFault(memstatus) then
        HandleExternalReadAbort(memstatus, memaddrdesc, size, accdesc);
    return value;

// AArch32.MemSingle[] - assignment (write) form
// =============================================

AArch32.MemSingle[bits(32) address, integer size,
                  AccessDescriptor accdesc, boolean aligned] = bits(size*8) value
    boolean ispair = FALSE;
    AArch32.MemSingle[address, size, accdesc, aligned, ispair] = value;
    return;

// AArch32.MemSingle[] - assignment (write) form
// =============================================
// Perform an atomic, little-endian write of 'size' bytes.

AArch32.MemSingle[bits(32) address, integer size, AccessDescriptor accdesc_in,
                  boolean aligned, boolean ispair] = bits(size*8) value
    assert size IN {1, 2, 4, 8, 16};
    AccessDescriptor accdesc = accdesc_in;
    assert IsAligned(address, size);

    AddressDescriptor memaddrdesc;
    memaddrdesc = AArch32.TranslateAddress(address, accdesc, aligned, size);

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        AArch32.Abort(address, memaddrdesc.fault);

    // Effect on exclusives
    if memaddrdesc.memattrs.shareability != Shareability_NSH then
        ClearExclusiveByAddress(memaddrdesc.paddress, ProcessorID(), size);

    if SPESampleInFlight then
        boolean is_load = FALSE;
```

```
        SPESampleLoadStore(is_load, accdesc, memaddrdesc);

    PhysMemRetStatus memstatus;
    memstatus = PhysMemWrite(memaddrdesc, size, accdesc, value);
    if IsFault(memstatus) then
        HandleExternalWriteAbort(memstatus, memaddrdesc, size, accdesc);
    return;
```

Library pseudocode for aarch32/functions/memory/AArch32.UnalignedAccessFaults

```
// AArch32.UnalignedAccessFaults()
// ===============================
// Determine whether the unaligned access generates an Alignment fault

boolean AArch32.UnalignedAccessFaults(AccessDescriptor accdesc)
    return (AlignmentEnforced() ||
            accdesc.a32lsmd     ||
            accdesc.exclusive   ||
            accdesc.acqsc       ||
            accdesc.relsc);
```

Library pseudocode for aarch32/functions/memory/Hint_PreloadData

```
// Hint_PreloadData()
// ==================

Hint_PreloadData(bits(32) address);
```

Library pseudocode for aarch32/functions/memory/Hint_PreloadDataForWrite

```
// Hint_PreloadDataForWrite()
// ==========================

Hint_PreloadDataForWrite(bits(32) address);
```

Library pseudocode for aarch32/functions/memory/Hint_PreloadInstr

```
// Hint_PreloadInstr()
// ===================

Hint_PreloadInstr(bits(32) address);
```

Library pseudocode for aarch32/functions/memory/MemA

```
// MemA[] - non-assignment form
// ============================

bits(8*size) MemA[bits(32) address, integer size]
    boolean acqrel = FALSE;
    boolean tagchecked = FALSE;
    AccessDescriptor accdesc = CreateAccDescExLDST(MemOp_LOAD, acqrel, tagchecked);
    return Mem_with_type[address, size, accdesc];

// MemA[] - assignment form
// ========================

MemA[bits(32) address, integer size] = bits(8*size) value
    boolean acqrel = FALSE;
    boolean tagchecked = FALSE;
    AccessDescriptor accdesc = CreateAccDescExLDST(MemOp_STORE, acqrel, tagchecked);
    Mem_with_type[address, size, accdesc] = value;
    return;
```

Library pseudocode for aarch32/functions/memory/MemO

```
// MemO[] - non-assignment form
// ============================

bits(8*size) MemO[bits(32) address, integer size]
    boolean tagchecked = FALSE;
    AccessDescriptor accdesc = CreateAccDescAcqRel(MemOp_LOAD, tagchecked);
    return Mem_with_type[address, size, accdesc];

// MemO[] - assignment form
// ========================

MemO[bits(32) address, integer size] = bits(8*size) value
    boolean tagchecked = FALSE;
    AccessDescriptor accdesc = CreateAccDescAcqRel(MemOp_STORE, tagchecked);
    Mem_with_type[address, size, accdesc] = value;
    return;
```

Library pseudocode for aarch32/functions/memory/MemS

```
// MemS[] - non-assignment form
// ============================
// Memory accessor for streaming load multiple instructions

bits(8*size) MemS[bits(32) address, integer size]
    AccessDescriptor accdesc = CreateAccDescA32LSMD(MemOp_LOAD);
    return Mem_with_type[address, size, accdesc];

// MemS[] - assignment form
// ========================
// Memory accessor for streaming store multiple instructions

MemS[bits(32) address, integer size] = bits(8*size) value
    AccessDescriptor accdesc = CreateAccDescA32LSMD(MemOp_STORE);
    Mem_with_type[address, size, accdesc] = value;
    return;
```

Library pseudocode for aarch32/functions/memory/MemU

```
// MemU[] - non-assignment form
// ============================

bits(8*size) MemU[bits(32) address, integer size]
    boolean nontemporal = FALSE;
    boolean privileged = PSTATE.EL != EL0;
    boolean tagchecked = FALSE;
    AccessDescriptor accdesc = CreateAccDescGPR(MemOp_LOAD, nontemporal, privileged, tagchecked);
    return Mem_with_type[address, size, accdesc];

// MemU[] - assignment form
// ========================

MemU[bits(32) address, integer size] = bits(8*size) value
    boolean nontemporal = FALSE;
    boolean privileged = PSTATE.EL != EL0;
    boolean tagchecked = FALSE;
    AccessDescriptor accdesc = CreateAccDescGPR(MemOp_STORE, nontemporal, privileged, tagchecked);
    Mem_with_type[address, size, accdesc] = value;
    return;
```

Library pseudocode for aarch32/functions/memory/MemU_unpriv

```
// MemU_unpriv[] - non-assignment form
// ===================================

bits(8*size) MemU_unpriv[bits(32) address, integer size]
    boolean nontemporal = FALSE;
    boolean privileged = FALSE;
    boolean tagchecked = FALSE;
    AccessDescriptor accdesc = CreateAccDescGPR(MemOp_LOAD, nontemporal, privileged, tagchecked);
    return Mem_with_type[address, size, accdesc];

// MemU_unpriv[] - assignment form
// ===============================

MemU_unpriv[bits(32) address, integer size] = bits(8*size) value
    boolean nontemporal = FALSE;
    boolean privileged = FALSE;
    boolean tagchecked = FALSE;
    AccessDescriptor accdesc = CreateAccDescGPR(MemOp_STORE, nontemporal, privileged, tagchecked);
    Mem_with_type[address, size, accdesc] = value;
    return;
```

Library pseudocode for aarch32/functions/memory/Mem_with_type

```
// Mem_with_type[] - non-assignment (read) form
// ===============================================
// Perform a read of 'size' bytes. The access byte order is reversed for a big-endian access.
// Instruction fetches would call AArch32.MemSingle directly.

bits(size*8) Mem_with_type[bits(32) address, integer size, AccessDescriptor accdesc]
    boolean ispair = FALSE;
    return Mem_with_type[address, size, accdesc, ispair];

bits(size*8) Mem_with_type[bits(32) address, integer size, AccessDescriptor accdesc, boolean ispair]
    assert size IN {1, 2, 4, 8, 16};
    constant halfsize = size DIV 2;
    bits(size * 8) value;

    // Check alignment on size of element accessed, not overall access size
    integer alignment = if ispair then halfsize else size;
    boolean aligned   = IsAligned(address, alignment);

    if !aligned && AArch32.UnalignedAccessFaults(accdesc) then
        AArch32.Abort(address, AlignmentFault(accdesc));

    if aligned then
        value = AArch32.MemSingle[address, size, accdesc, aligned, ispair];
    else
        assert size > 1;
        value<7:0> = AArch32.MemSingle[address, 1, accdesc, aligned];

        // For subsequent bytes it is CONSTRAINED UNPREDICTABLE whether an unaligned Device memory
        // access will generate an Alignment Fault, as to get this far means the first byte did
        // not, so we must be changing to a new translation page.
        c = ConstrainUnpredictable(Unpredictable_DEVPAGE2);
        assert c IN {Constraint_FAULT, Constraint_NONE};
        if c == Constraint_NONE then aligned = TRUE;

        for i = 1 to size-1
            value<8*i+7:8*i> = AArch32.MemSingle[address+i, 1, accdesc, aligned];

    if BigEndian(accdesc.acctype) then
        value = BigEndianReverse(value);

    return value;

// Mem_with_type[] - assignment (write) form
// =========================================
// Perform a write of 'size' bytes. The byte order is reversed for a big-endian access.

Mem_with_type[bits(32) address, integer size, AccessDescriptor accdesc] = bits(size*8) value_in
    boolean ispair = FALSE;
    Mem_with_type[address, size, accdesc, ispair] = value_in;

Mem_with_type[bits(32) address, integer size, AccessDescriptor accdesc,
        boolean ispair] = bits(size*8) value_in
    constant halfsize = size DIV 2;
    bits(size*8) value = value_in;

    // Check alignment on size of element accessed, not overall access size
    integer alignment = if ispair then halfsize else size;
    boolean aligned   = IsAligned(address, alignment);

    if !aligned && AArch32.UnalignedAccessFaults(accdesc) then
        AArch32.Abort(address, AlignmentFault(accdesc));

    if BigEndian(accdesc.acctype) then
        value = BigEndianReverse(value);

    if aligned then
        AArch32.MemSingle[address, size, accdesc, aligned, ispair] = value;
    else
        assert size > 1;
        AArch32.MemSingle[address, 1, accdesc, aligned] = value<7:0>;
```

```
        // For subsequent bytes it is CONSTRAINED UNPREDICTABLE whether an unaligned Device memory
        // access will generate an Alignment Fault, as to get this far means the first byte did
        // not, so we must be changing to a new translation page.

        c = ConstrainUnpredictable(Unpredictable_DEVPAGE2);
        assert c IN {Constraint_FAULT, Constraint_NONE};
        if c == Constraint_NONE then aligned = TRUE;

        for i = 1 to size-1
            AArch32.MemSingle[address+i, 1, accdesc, aligned] = value<8*i+7:8*i>;
    return;
```

Library pseudocode for aarch32/functions/ras/AArch32.ESBOperation

```
// AArch32.ESBOperation()
// ======================
// Perform the AArch32 ESB operation for ESB executed in AArch32 state

AArch32.ESBOperation()

    // Check if routed to AArch64 state
    route_to_aarch64 = PSTATE.EL == EL0 && !ELUsingAArch32(EL1);
    if !route_to_aarch64 && EL2Enabled() && !ELUsingAArch32(EL2) then
        route_to_aarch64 = HCR_EL2.TGE == '1' || HCR_EL2.AMO == '1';
    if !route_to_aarch64 && HaveEL(EL3) && !ELUsingAArch32(EL3) then
        route_to_aarch64 = EffectiveEA() == '1';

    if route_to_aarch64 then
        AArch64.ESBOperation();
        return;

    route_to_monitor = HaveEL(EL3) && ELUsingAArch32(EL3) && EffectiveEA() == '1';
    route_to_hyp = PSTATE.EL IN {EL0, EL1} && EL2Enabled() && (HCR.TGE == '1' || HCR.AMO == '1');

    bits(5) target;
    if route_to_monitor then
        target = M32_Monitor;
    elsif route_to_hyp || PSTATE.M == M32_Hyp then
        target = M32_Hyp;
    else
        target = M32_Abort;

    boolean mask_active;
    if CurrentSecurityState() == SS_Secure then
        mask_active = TRUE;
    elsif target == M32_Monitor then
        mask_active = SCR.AW == '1' && (!HaveEL(EL2) || (HCR.TGE == '0' && HCR.AMO == '0'));
    else
        mask_active = target == M32_Abort || PSTATE.M == M32_Hyp;

    mask_set = PSTATE.A == '1';
    (-, el)  = ELFromM32(target);
    intdis   = Halted() || ExternalDebugInterruptsDisabled(el);
    masked   = intdis || (mask_active && mask_set);

    // Check for a masked Physical SError pending that can be synchronized
    // by an Error synchronization event.
    if masked && IsSynchronizablePhysicalSErrorPending() then
        bits(32) syndrome = Zeros(32);
        syndrome<31> = '1';  // A
        syndrome<15:0> = AArch32.PhysicalSErrorSyndrome();
        DISR = syndrome;
        ClearPendingPhysicalSError();

    return;
```

Library pseudocode for aarch32/functions/ras/AArch32.EncodeAsyncErrorSyndrome

```
// AArch32.EncodeAsyncErrorSyndrome()
// ==================================
// Return the corresponding encoding for ErrorState.

bits(2) AArch32.EncodeAsyncErrorSyndrome(ErrorState errorstate)
    case errorstate of
        when ErrorState_UC   return '00';
        when ErrorState_UEU  return '01';
        when ErrorState_UEO  return '10';
        when ErrorState_UER  return '11';
        otherwise Unreachable();
```

Library pseudocode for aarch32/functions/ras/AArch32.PhysicalSErrorSyndrome

```
// AArch32.PhysicalSErrorSyndrome()
// ================================
// Generate SError syndrome.

bits(16) AArch32.PhysicalSErrorSyndrome()
    bits(32) syndrome = Zeros(32);
    FaultRecord fault = GetPendingPhysicalSError();
    if PSTATE.EL == EL2 then
        ErrorState errstate = AArch32.PEErrorState(fault);
        syndrome<11:10> = AArch32.EncodeAsyncErrorSyndrome(errstate);   // AET
        syndrome<9>     = fault.extflag;                                 // EA
        syndrome<5:0>   = '010001';                                      // DFSC
    else
        boolean long_format = TTBCR.EAE == '1';
        syndrome = AArch32.CommonFaultStatus(fault, long_format);
    return syndrome<15:0>;
```

Library pseudocode for aarch32/functions/ras/AArch32.vESBOperation

```
// AArch32.vESBOperation()
// =======================
// Perform the ESB operation for virtual SError interrupts executed in AArch32 state

AArch32.vESBOperation()
    assert PSTATE.EL IN {EL0, EL1} && EL2Enabled();

    // Check for EL2 using AArch64 state
    if !ELUsingAArch32(EL2) then
        AArch64.vESBOperation();
        return;

    // If physical SError interrupts are routed to Hyp mode, and TGE is not set,
    // then a virtual SError interrupt might be pending
    vSEI_enabled = HCR.TGE == '0' && HCR.AMO == '1';
    vSEI_pending = vSEI_enabled && HCR.VA == '1';
    vintdis      = Halted() || ExternalDebugInterruptsDisabled(EL1);
    vmasked      = vintdis || PSTATE.A == '1';

    // Check for a masked virtual SError pending
    if vSEI_pending && vmasked then
        bits(32) syndrome = Zeros(32);
        syndrome<31> = '1';               // A
        syndrome<15:14> = VDFSR<15:14>;  // AET
        syndrome<12>    = VDFSR<12>;     // ExT
        syndrome<9>     = TTBCR.EAE;     // LPAE
        if TTBCR.EAE == '1' then         // Long-descriptor format
            syndrome<5:0>    = '010001'; // STATUS
        else                             // Short-descriptor format
            syndrome<10,3:0> = '10110';  // FS
        VDISR = syndrome;
        HCR.VA = '0';                    // Clear pending virtual SError

    return;
```

Library pseudocode for aarch32/functions/registers/AArch32.ResetGeneralRegisters

```
// AArch32.ResetGeneralRegisters()
// ===============================

AArch32.ResetGeneralRegisters()

    for i = 0 to 7
        R[i] = bits(32) UNKNOWN;
    for i = 8 to 12
        Rmode[i, M32_User] = bits(32) UNKNOWN;
        Rmode[i, M32_FIQ] = bits(32) UNKNOWN;
    if HaveEL(EL2) then Rmode[13, M32_Hyp] = bits(32) UNKNOWN;   // No R14_hyp
    for i = 13 to 14
        Rmode[i, M32_User] = bits(32) UNKNOWN;
        Rmode[i, M32_FIQ] = bits(32) UNKNOWN;
        Rmode[i, M32_IRQ] = bits(32) UNKNOWN;
        Rmode[i, M32_Svc] = bits(32) UNKNOWN;
        Rmode[i, M32_Abort] = bits(32) UNKNOWN;
        Rmode[i, M32_Undef] = bits(32) UNKNOWN;
        if HaveEL(EL3) then Rmode[i, M32_Monitor] = bits(32) UNKNOWN;

    return;
```

Library pseudocode for aarch32/functions/registers/AArch32.ResetSIMDFPRegisters

```
// AArch32.ResetSIMDFPRegisters()
// ==============================

AArch32.ResetSIMDFPRegisters()

    for i = 0 to 15
        Q[i] = bits(128) UNKNOWN;

    return;
```

Library pseudocode for aarch32/functions/registers/AArch32.ResetSpecialRegisters

```
// AArch32.ResetSpecialRegisters()
// ===============================

AArch32.ResetSpecialRegisters()

    // AArch32 special registers
    SPSR_fiq<31:0> = bits(32) UNKNOWN;
    SPSR_irq<31:0> = bits(32) UNKNOWN;
    SPSR_svc<31:0> = bits(32) UNKNOWN;
    SPSR_abt<31:0> = bits(32) UNKNOWN;
    SPSR_und<31:0> = bits(32) UNKNOWN;
    if HaveEL(EL2) then
        SPSR_hyp = bits(32) UNKNOWN;
        ELR_hyp = bits(32) UNKNOWN;
    if HaveEL(EL3) then
        SPSR_mon = bits(32) UNKNOWN;

    // External debug special registers
    DLR = bits(32) UNKNOWN;
    DSPSR = bits(32) UNKNOWN;

    return;
```

Library pseudocode for aarch32/functions/registers/AArch32.ResetSystemRegisters

```
// AArch32.ResetSystemRegisters()
// ==============================

AArch32.ResetSystemRegisters(boolean cold_reset);
```

Library pseudocode for aarch32/functions/registers/ALUExceptionReturn

```
// ALUExceptionReturn()
// ====================

ALUExceptionReturn(bits(32) address)
    if PSTATE.EL == EL2 then
        UNDEFINED;
    elsif PSTATE.M IN {M32_User,M32_System} then
        Constraint c = ConstrainUnpredictable(Unpredictable_ALUEXCEPTIONRETURN);
        assert c IN {Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_UNDEF
                UNDEFINED;
            when Constraint_NOP
                EndOfInstruction();
    else
        AArch32.ExceptionReturn(address, SPSR[]);
```

Library pseudocode for aarch32/functions/registers/ALUWritePC

```
// ALUWritePC()
// ============

ALUWritePC(bits(32) address)
    if CurrentInstrSet() == InstrSet_A32 then
        BXWritePC(address, BranchType_INDIR);
    else
        BranchWritePC(address, BranchType_INDIR);
```

Library pseudocode for aarch32/functions/registers/BXWritePC

```
// BXWritePC()
// ===========

BXWritePC(bits(32) address_in, BranchType branch_type)
    bits(32) address = address_in;
    if address<0> == '1' then
        SelectInstrSet(InstrSet_T32);
        address<0> = '0';
    else
        SelectInstrSet(InstrSet_A32);
        // For branches to an unaligned PC counter in A32 state, the processor takes the branch
        // and does one of:
        // * Forces the address to be aligned
        // * Leaves the PC unaligned, meaning the target generates a PC Alignment fault.
        if address<1> == '1' && ConstrainUnpredictableBool(Unpredictable_A32FORCEALIGNPC) then
            address<1> = '0';
    boolean branch_conditional = !(AArch32.CurrentCond() IN {'111x'});
    BranchTo(address, branch_type, branch_conditional);
```

Library pseudocode for aarch32/functions/registers/BranchWritePC

```
// BranchWritePC()
// ===============

BranchWritePC(bits(32) address_in, BranchType branch_type)
    bits(32) address = address_in;
    if CurrentInstrSet() == InstrSet_A32 then
        address<1:0> = '00';
    else
        address<0> = '0';
    boolean branch_conditional = !(AArch32.CurrentCond() IN {'111x'});
    BranchTo(address, branch_type, branch_conditional);
```

Library pseudocode for aarch32/functions/registers/CBWritePC

```
// CBWritePC()
// ===========
// Takes a branch from a CBNZ/CBZ instruction.

CBWritePC(bits(32) address_in)
    bits(32) address = address_in;
    assert CurrentInstrSet() == InstrSet_T32;
    address<0> = '0';
    boolean branch_conditional = TRUE;
    BranchTo(address, BranchType_DIR, branch_conditional);
```

Library pseudocode for aarch32/functions/registers/D

```
// D[] - non-assignment form
// =========================

bits(64) D[integer n]
    assert n >= 0 && n <= 31;
    base = (n MOD 2) * 64;
    bits(128) vreg = V[n DIV 2, 128];
    return vreg<base+63:base>;

// D[] - assignment form
// =====================

D[integer n] = bits(64) value
    assert n >= 0 && n <= 31;
    base = (n MOD 2) * 64;
    bits(128) vreg = V[n DIV 2, 128];
    vreg<base+63:base> = value;
    V[n DIV 2, 128] = vreg;
    return;
```

Library pseudocode for aarch32/functions/registers/Din

```
// Din[] - non-assignment form
// ===========================

bits(64) Din[integer n]
    assert n >= 0 && n <= 31;
    return _Dclone[n];
```

Library pseudocode for aarch32/functions/registers/LR

```
// LR - assignment form
// ====================

LR = bits(32) value
    R[14] = value;
    return;

// LR - non-assignment form
// ========================

bits(32) LR
    return R[14];
```

Library pseudocode for aarch32/functions/registers/LoadWritePC

```
// LoadWritePC()
// =============

LoadWritePC(bits(32) address)
    BXWritePC(address, BranchType_INDIR);
```

Library pseudocode for aarch32/functions/registers/LookUpRIndex

```
// LookUpRIndex()
// ==============

integer LookUpRIndex(integer n, bits(5) mode)
    assert n >= 0 && n <= 14;

    integer result;
    case n of  // Select  index by mode:     usr fiq irq svc abt und hyp
        when 8     result = RBankSelect(mode,  8, 24,  8,  8,  8,  8,  8);
        when 9     result = RBankSelect(mode,  9, 25,  9,  9,  9,  9,  9);
        when 10    result = RBankSelect(mode, 10, 26, 10, 10, 10, 10, 10);
        when 11    result = RBankSelect(mode, 11, 27, 11, 11, 11, 11, 11);
        when 12    result = RBankSelect(mode, 12, 28, 12, 12, 12, 12, 12);
        when 13    result = RBankSelect(mode, 13, 29, 17, 19, 21, 23, 15);
        when 14    result = RBankSelect(mode, 14, 30, 16, 18, 20, 22, 14);
        otherwise  result = n;

    return result;
```

Library pseudocode for aarch32/functions/registers/Monitor_mode_registers

```
bits(32) SP_mon;
bits(32) LR_mon;
```

Library pseudocode for aarch32/functions/registers/PC

```
// PC - non-assignment form
// ========================

bits(32) PC
    return R[15];               // This includes the offset from AArch32 state
```

Library pseudocode for aarch32/functions/registers/PCStoreValue

```
// PCStoreValue()
// ==============

bits(32) PCStoreValue()
    // This function returns the PC value. On architecture versions before Armv7, it
    // is permitted to instead return PC+4, provided it does so consistently. It is
    // used only to describe A32 instructions, so it returns the address of the current
    // instruction plus 8 (normally) or 12 (when the alternative is permitted).
    return PC;
```

Library pseudocode for aarch32/functions/registers/Q

```
// Q[] - non-assignment form
// =========================

bits(128) Q[integer n]
    assert n >= 0 && n <= 15;
    return V[n, 128];

// Q[] - assignment form
// =====================

Q[integer n] = bits(128) value
    assert n >= 0 && n <= 15;
    V[n, 128] = value;
    return;
```

Library pseudocode for aarch32/functions/registers/Qin

```
// Qin[] - non-assignment form
// ===========================

bits(128) Qin[integer n]
    assert n >= 0 && n <= 15;
    return Din[2*n+1]:Din[2*n];
```

Library pseudocode for aarch32/functions/registers/R

```
// R[] - assignment form
// =====================

R[integer n] = bits(32) value
    Rmode[n, PSTATE.M] = value;
    return;

// R[] - non-assignment form
// =========================

bits(32) R[integer n]
    if n == 15 then
        offset = (if CurrentInstrSet() == InstrSet_A32 then 8 else 4);
        return _PC<31:0> + offset;
    else
        return Rmode[n, PSTATE.M];
```

Library pseudocode for aarch32/functions/registers/RBankSelect

```
// RBankSelect()
// =============

integer RBankSelect(bits(5) mode, integer usr, integer fiq, integer irq,
                    integer svc, integer abt, integer und, integer hyp)

    integer result;
    case mode of
        when M32_User    result = usr;  // User mode
        when M32_FIQ     result = fiq;  // FIQ mode
        when M32_IRQ     result = irq;  // IRQ mode
        when M32_Svc     result = svc;  // Supervisor mode
        when M32_Abort   result = abt;  // Abort mode
        when M32_Hyp     result = hyp;  // Hyp mode
        when M32_Undef   result = und;  // Undefined mode
        when M32_System  result = usr;  // System mode uses User mode registers
        otherwise        Unreachable(); // Monitor mode

    return result;
```

Library pseudocode for aarch32/functions/registers/Rmode

```
// Rmode[] - non-assignment form
// =============================

bits(32) Rmode[integer n, bits(5) mode]
    assert n >= 0 && n <= 14;

    // Check for attempted use of Monitor mode in Non-secure state.
    if CurrentSecurityState() != SS_Secure then assert mode != M32_Monitor;
    assert !BadMode(mode);

    if mode == M32_Monitor then
        if n == 13 then return SP_mon;
        elsif n == 14 then return LR_mon;
        else return _R[n]<31:0>;
    else
        return _R[LookUpRIndex(n, mode)]<31:0>;

// Rmode[] - assignment form
// =========================

Rmode[integer n, bits(5) mode] = bits(32) value
    assert n >= 0 && n <= 14;

    // Check for attempted use of Monitor mode in Non-secure state.
    if CurrentSecurityState() != SS_Secure then assert mode != M32_Monitor;
    assert !BadMode(mode);

    if mode == M32_Monitor then
        if n == 13 then SP_mon = value;
        elsif n == 14 then LR_mon = value;
        else _R[n]<31:0> = value;
    else
        // It is CONSTRAINED UNPREDICTABLE whether the upper 32 bits of the X
        // register are unchanged or set to zero. This is also tested for on
        // exception entry, as this applies to all AArch32 registers.
        if HaveAArch64() && ConstrainUnpredictableBool(Unpredictable_ZEROUPPER) then
            _R[LookUpRIndex(n, mode)] = ZeroExtend(value, 64);
        else
            _R[LookUpRIndex(n, mode)]<31:0> = value;

    return;
```

Library pseudocode for aarch32/functions/registers/S

```
// S[] - non-assignment form
// =========================

bits(32) S[integer n]
    assert n >= 0 && n <= 31;
    base = (n MOD 4) * 32;
    bits(128) vreg = V[n DIV 4, 128];
    return vreg<base+31:base>;

// S[] - assignment form
// =====================

S[integer n] = bits(32) value
    assert n >= 0 && n <= 31;
    base = (n MOD 4) * 32;
    bits(128) vreg = V[n DIV 4, 128];
    vreg<base+31:base> = value;
    V[n DIV 4, 128] = vreg;
    return;
```

Library pseudocode for aarch32/functions/registers/_Dclone

```
// _Dclone[]
// =========
// Clone the 64-bit Advanced SIMD and VFP extension register bank for use as input to
// instruction pseudocode, to avoid read-after-write for Advanced SIMD and VFP operations.

array bits(64) _Dclone[0..31];
```

Library pseudocode for aarch32/functions/system/AArch32.ExceptionReturn

```
// AArch32.ExceptionReturn()
// =========================

AArch32.ExceptionReturn(bits(32) new_pc_in, bits(32) spsr)
    bits(32) new_pc = new_pc_in;
    SynchronizeContext();
    // Attempts to change to an illegal mode or state will invoke the Illegal Execution state
    // mechanism
    SetPSTATEFromPSR(spsr);
    ClearExclusiveLocal(ProcessorID());
    SendEventLocal();

    if PSTATE.IL == '1' then
        // If the exception return is illegal, PC[1:0] are UNKNOWN
        new_pc<1:0> = bits(2) UNKNOWN;
    else
        // LR[1:0] or LR[0] are treated as being 0, depending on the target instruction set state
        if PSTATE.T == '1' then
            new_pc<0> = '0';                 // T32
        else
            new_pc<1:0> = '00';              // A32

    boolean branch_conditional = !(AArch32.CurrentCond() IN {'111x'});
    BranchTo(new_pc, BranchType_ERET, branch_conditional);

    CheckExceptionCatch(FALSE);              // Check for debug event on exception return
```

Library pseudocode for aarch32/functions/system/AArch32.ExecutingCP10or11Instr

```
// AArch32.ExecutingCP10or11Instr()
// ================================

boolean AArch32.ExecutingCP10or11Instr()
    instr =  ThisInstr();
    instr_set = CurrentInstrSet();
    assert instr_set IN {InstrSet_A32, InstrSet_T32};

    if instr_set == InstrSet_A32 then
        return ((instr<27:24> == '1110' || instr<27:25> == '110') && instr<11:8> IN {'101x'});
    else // InstrSet_T32
        return (instr<31:28> IN {'111x'} && (instr<27:24> == '1110' || instr<27:25> == '110') &&
                instr<11:8> IN {'101x'});
```

Library pseudocode for aarch32/functions/system/AArch32.ITAdvance

```
// AArch32.ITAdvance()
// ===================

AArch32.ITAdvance()
    if PSTATE.IT<2:0> == '000' then
        PSTATE.IT = '00000000';
    else
        PSTATE.IT<4:0> = LSL(PSTATE.IT<4:0>, 1);
    return;
```

Library pseudocode for aarch32/functions/system/AArch32.SysRegRead

```
// AArch32.SysRegRead()
// ====================
// Read from a 32-bit AArch32 System register and write the register's contents to R[t].

AArch32.SysRegRead(integer cp_num, bits(32) instr, integer t);
```

Library pseudocode for aarch32/functions/system/AArch32.SysRegRead64

```
// AArch32.SysRegRead64()
// ======================
// Read from a 64-bit AArch32 System register and write the register's contents to R[t] and R[t2].

AArch32.SysRegRead64(integer cp_num, bits(32) instr, integer t, integer t2);
```

Library pseudocode for aarch32/functions/system/AArch32.SysRegReadCanWriteAPSR

```
// AArch32.SysRegReadCanWriteAPSR()
// ================================
// Determines whether the AArch32 System register read instruction can write to APSR flags.

boolean AArch32.SysRegReadCanWriteAPSR(integer cp_num, bits(32) instr)
    assert UsingAArch32();
    assert (cp_num IN {14,15});
    assert cp_num == UInt(instr<11:8>);

    opc1 = UInt(instr<23:21>);
    opc2 = UInt(instr<7:5>);
    CRn  = UInt(instr<19:16>);
    CRm  = UInt(instr<3:0>);

    if cp_num == 14 && opc1 == 0 && CRn == 0 && CRm == 1 && opc2 == 0 then // DBGDSCRint
        return TRUE;

    return FALSE;
```

Library pseudocode for aarch32/functions/system/AArch32.SysRegWrite

```
// AArch32.SysRegWrite()
// =====================
// Read the contents of R[t] and write to a 32-bit AArch32 System register.

AArch32.SysRegWrite(integer cp_num, bits(32) instr, integer t);
```

Library pseudocode for aarch32/functions/system/AArch32.SysRegWrite64

```
// AArch32.SysRegWrite64()
// =======================
// Read the contents of R[t] and R[t2] and write to a 64-bit AArch32 System register.

AArch32.SysRegWrite64(integer cp_num, bits(32) instr, integer t, integer t2);
```

Library pseudocode for aarch32/functions/system/AArch32.SysRegWriteM

```
// AArch32.SysRegWriteM()
// ======================
// Read a value from a virtual address and write it to an AArch32 System register.

AArch32.SysRegWriteM(integer cp_num, bits(32) instr, bits(32) address);
```

Library pseudocode for aarch32/functions/system/AArch32.WriteMode

```
// AArch32.WriteMode()
// ===================
// Function for dealing with writes to PSTATE.M from AArch32 state only.
// This ensures that PSTATE.EL and PSTATE.SP are always valid.

AArch32.WriteMode(bits(5) mode)
    (valid,el) = ELFromM32(mode);
    assert valid;
    PSTATE.M   = mode;
    PSTATE.EL  = el;
    PSTATE.nRW = '1';
    PSTATE.SP  = (if mode IN {M32_User,M32_System} then '0' else '1');
    return;
```

Library pseudocode for aarch32/functions/system/AArch32.WriteModeByInstr

```
// AArch32.WriteModeByInstr()
// ==========================
// Function for dealing with writes to PSTATE.M from an AArch32 instruction, and ensuring that
// illegal state changes are correctly flagged in PSTATE.IL.

AArch32.WriteModeByInstr(bits(5) mode)
    (valid,el) = ELFromM32(mode);

    // 'valid' is set to FALSE if' mode' is invalid for this implementation or the current value
    // of SCR.NS/SCR_EL3.NS. Additionally, it is illegal for an instruction to write 'mode' to
    // PSTATE.EL if it would result in any of:
    // * A change to a mode that would cause entry to a higher Exception level.
    if UInt(el) > UInt(PSTATE.EL) then
        valid = FALSE;

    // * A change to or from Hyp mode.
    if (PSTATE.M == M32_Hyp || mode == M32_Hyp) && PSTATE.M != mode then
        valid = FALSE;

    // * When EL2 is implemented, the value of HCR.TGE is '1', a change to a Non-secure EL1 mode.
    if PSTATE.M == M32_Monitor && HaveEL(EL2) && el == EL1 && SCR.NS == '1' && HCR.TGE == '1' then
        valid = FALSE;

    if !valid then
        PSTATE.IL = '1';
    else
        AArch32.WriteMode(mode);
```

Library pseudocode for aarch32/functions/system/BadMode

```
// BadMode()
// =========

boolean BadMode(bits(5) mode)
    // Return TRUE if 'mode' encodes a mode that is not valid for this implementation
    boolean valid;
    case mode of
        when M32_Monitor
            valid = HaveAArch32EL(EL3);
        when M32_Hyp
            valid = HaveAArch32EL(EL2);
        when M32_FIQ, M32_IRQ, M32_Svc, M32_Abort, M32_Undef, M32_System
            // If EL3 is implemented and using AArch32, then these modes are EL3 modes in Secure
            // state, and EL1 modes in Non-secure state. If EL3 is not implemented or is using
            // AArch64, then these modes are EL1 modes.
            // Therefore it is sufficient to test this implementation supports EL1 using AArch32.
            valid = HaveAArch32EL(EL1);
        when M32_User
            valid = HaveAArch32EL(EL0);
        otherwise
            valid = FALSE;           // Passed an illegal mode value
    return !valid;
```

Library pseudocode for aarch32/functions/system/BankedRegisterAccessValid

```
// BankedRegisterAccessValid()
// ===========================
// Checks for MRS (Banked register) or MSR (Banked register) accesses to registers
// other than the SPSRs that are invalid. This includes ELR_hyp accesses.

BankedRegisterAccessValid(bits(5) SYSm, bits(5) mode)

    case SYSm of
        when '000xx', '00100'                          // R8_usr to R12_usr
            if mode != M32_FIQ then UNPREDICTABLE;
        when '00101'                                   // SP_usr
            if mode == M32_System then UNPREDICTABLE;
        when '00110'                                   // LR_usr
            if mode IN {M32_Hyp,M32_System} then UNPREDICTABLE;
        when '010xx', '0110x', '01110'                 // R8_fiq to R12_fiq, SP_fiq, LR_fiq
            if mode == M32_FIQ then UNPREDICTABLE;
        when '1000x'                                   // LR_irq, SP_irq
            if mode == M32_IRQ then UNPREDICTABLE;
        when '1001x'                                   // LR_svc, SP_svc
            if mode == M32_Svc then UNPREDICTABLE;
        when '1010x'                                   // LR_abt, SP_abt
            if mode == M32_Abort then UNPREDICTABLE;
        when '1011x'                                   // LR_und, SP_und
            if mode == M32_Undef then UNPREDICTABLE;
        when '1110x'                                   // LR_mon, SP_mon
            if (!HaveEL(EL3) || CurrentSecurityState() != SS_Secure ||
                mode == M32_Monitor) then UNPREDICTABLE;
        when '11110'                                   // ELR_hyp, only from Monitor or Hyp mode
            if !HaveEL(EL2) || !(mode IN {M32_Monitor,M32_Hyp}) then UNPREDICTABLE;
        when '11111'                                   // SP_hyp, only from Monitor mode
            if !HaveEL(EL2) || mode != M32_Monitor then UNPREDICTABLE;
        otherwise
            UNPREDICTABLE;

    return;
```

Library pseudocode for aarch32/functions/system/CPSRWriteByInstr

```
// CPSRWriteByInstr()
// ==================
// Update PSTATE.<N,Z,C,V,Q,GE,E,A,I,F,M> from a CPSR value written by an MSR instruction.

CPSRWriteByInstr(bits(32) value, bits(4) bytemask)
    privileged = PSTATE.EL != EL0;              // PSTATE.<A,I,F,M> are not writable at EL0

    // Write PSTATE from 'value', ignoring bytes masked by 'bytemask'
    if bytemask<3> == '1' then
        PSTATE.<N,Z,C,V,Q> = value<31:27>;
        // Bits <26:24> are ignored

    if bytemask<2> == '1' then
        if HaveSSBSExt() then
            PSTATE.SSBS = value<23>;
        if privileged then
            PSTATE.PAN = value<22>;
        if HaveDITExt() then
            PSTATE.DIT = value<21>;
        // Bit <20> is RES0
        PSTATE.GE = value<19:16>;

    if bytemask<1> == '1' then
        // Bits <15:10> are RES0
        PSTATE.E = value<9>;                    // PSTATE.E is writable at EL0
        if privileged then
            PSTATE.A = value<8>;

    if bytemask<0> == '1' then
        if privileged then
            PSTATE.<I,F> = value<7:6>;
            // Bit <5> is RES0
            // AArch32.WriteModeByInstr() sets PSTATE.IL to 1 if this is an illegal mode change.
            AArch32.WriteModeByInstr(value<4:0>);
    return;
```

Library pseudocode for aarch32/functions/system/ConditionPassed

```
// ConditionPassed()
// =================

boolean ConditionPassed()
    return ConditionHolds(AArch32.CurrentCond());
```

Library pseudocode for aarch32/functions/system/CurrentCond

```
// CurrentCond()
// =============

bits(4) AArch32.CurrentCond();
```

Library pseudocode for aarch32/functions/system/InITBlock

```
// InITBlock()
// ===========

boolean InITBlock()
    if CurrentInstrSet() == InstrSet_T32 then
        return PSTATE.IT<3:0> != '0000';
    else
        return FALSE;
```

Library pseudocode for aarch32/functions/system/LastInITBlock

```
// LastInITBlock()
// ===============

boolean LastInITBlock()
    return (PSTATE.IT<3:0> == '1000');
```

Library pseudocode for aarch32/functions/system/SPSRWriteByInstr

```
// SPSRWriteByInstr()
// ==================

SPSRWriteByInstr(bits(32) value, bits(4) bytemask)

    bits(32) new_spsr = SPSR[];

    if bytemask<3> == '1' then
        new_spsr<31:24> = value<31:24>;  // N,Z,C,V,Q flags, IT[1:0],J bits

    if bytemask<2> == '1' then
        new_spsr<23:16> = value<23:16>;  // IL bit, GE[3:0] flags

    if bytemask<1> == '1' then
        new_spsr<15:8> = value<15:8>;    // IT[7:2] bits, E bit, A interrupt mask

    if bytemask<0> == '1' then
        new_spsr<7:0> = value<7:0>;      // I,F interrupt masks, T bit, Mode bits

    SPSR[] = new_spsr;                   // UNPREDICTABLE if User or System mode

    return;
```

Library pseudocode for aarch32/functions/system/SPSRaccessValid

```
// SPSRaccessValid()
// =================
// Checks for MRS (Banked register) or MSR (Banked register) accesses to the SPSRs
// that are UNPREDICTABLE

SPSRaccessValid(bits(5) SYSm, bits(5) mode)
    case SYSm of
        when '01110'                                                    // SPSR_fiq
            if mode == M32_FIQ   then UNPREDICTABLE;
        when '10000'                                                    // SPSR_irq
            if mode == M32_IRQ   then UNPREDICTABLE;
        when '10010'                                                    // SPSR_svc
            if mode == M32_Svc   then UNPREDICTABLE;
        when '10100'                                                    // SPSR_abt
            if mode == M32_Abort then UNPREDICTABLE;
        when '10110'                                                    // SPSR_und
            if mode == M32_Undef then UNPREDICTABLE;
        when '11100'                                                    // SPSR_mon
            if (!HaveEL(EL3) || mode == M32_Monitor ||
                CurrentSecurityState() != SS_Secure) then UNPREDICTABLE;
        when '11110'                                                    // SPSR_hyp
            if !HaveEL(EL2) || mode != M32_Monitor then UNPREDICTABLE;
        otherwise
            UNPREDICTABLE;

    return;
```

Library pseudocode for aarch32/functions/system/SelectInstrSet

```
// SelectInstrSet()
// ================

SelectInstrSet(InstrSet iset)
    assert CurrentInstrSet() IN {InstrSet_A32, InstrSet_T32};
    assert iset IN {InstrSet_A32, InstrSet_T32};

    PSTATE.T = if iset == InstrSet_A32 then '0' else '1';

    return;
```

Library pseudocode for aarch32/functions/tlbi/AArch32.DTLBI_ALL

```
// AArch32.DTLBI_ALL()
// ===================
// Invalidate all data TLB entries for the indicated translation regime with the
// the indicated security state for all TLBs within the indicated shareability domain.
// Invalidation applies to all applicable stage 1 and stage 2 entries.

AArch32.DTLBI_ALL(SecurityState security, Regime regime, Shareability shareability,
                  TLBIMemAttr attr)
    assert PSTATE.EL IN {EL3, EL2, EL1};

    TLBIRecord r;
    r.op           = TLBIOp_DALL;
    r.from_aarch64 = FALSE;
    r.security     = security;
    r.regime       = regime;
    r.level        = TLBILevel_Any;
    r.attr         = attr;

    TLBI(r);
    if shareability != Shareability_NSH then Broadcast(shareability, r);
    return;
```

Library pseudocode for aarch32/functions/tlbi/AArch32.DTLBI_ASID

```
// AArch32.DTLBI_ASID()
// ====================
// Invalidate all data TLB stage 1 entries matching the indicated VMID (where regime supports)
// and ASID in the parameter Rt in the indicated translation regime with the
// indicated security state for all TLBs within the indicated shareability domain.
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.

AArch32.DTLBI_ASID(SecurityState security, Regime regime, bits(16) vmid,
                   Shareability shareability, TLBIMemAttr attr, bits(32) Rt)
    assert PSTATE.EL IN {EL3, EL2, EL1};

    TLBIRecord r;
    r.op           = TLBIOp_DASID;
    r.from_aarch64 = FALSE;
    r.security     = security;
    r.regime       = regime;
    r.vmid         = vmid;
    r.level        = TLBILevel_Any;
    r.attr         = attr;
    r.asid         = Zeros(8) : Rt<7:0>;

    TLBI(r);
    if shareability != Shareability_NSH then Broadcast(shareability, r);
    return;
```

Library pseudocode for aarch32/functions/tlbi/AArch32.DTLBI_VA

```
// AArch32.DTLBI_VA()
// ==================
// Invalidate by VA all stage 1 data TLB entries in the indicated shareability domain
// matching the indicated VMID and ASID (where regime supports VMID, ASID) in the indicated regime
// with the indicated security state.
// ASID, VA and related parameters are derived from Rt.
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.

AArch32.DTLBI_VA(SecurityState security, Regime regime, bits(16) vmid,
                 Shareability shareability, TLBILevel level,  TLBIMemAttr attr, bits(32) Rt)
    assert PSTATE.EL IN {EL3, EL2, EL1};

    TLBIRecord r;
    r.op           = TLBIOp_DVA;
    r.from_aarch64 = FALSE;
    r.security     = security;
    r.regime       = regime;
    r.vmid         = vmid;
    r.level        = level;
    r.attr         = attr;
    r.asid         = Zeros(8) : Rt<7:0>;
    r.address      = Zeros(32) : Rt<31:12> : Zeros(12);

    TLBI(r);
    if shareability != Shareability_NSH then Broadcast(shareability, r);
    return;
```

Library pseudocode for aarch32/functions/tlbi/AArch32.ITLBI_ALL

```
// AArch32.ITLBI_ALL()
// ===================
// Invalidate all instruction TLB entries for the indicated translation regime with the
// the indicated security state for all TLBs within the indicated shareability domain.
// Invalidation applies to all applicable stage 1 and stage 2 entries.

AArch32.ITLBI_ALL(SecurityState security, Regime regime, Shareability shareability,
                  TLBIMemAttr attr)
    assert PSTATE.EL IN {EL3, EL2, EL1};

    TLBIRecord r;
    r.op           = TLBIOp_IALL;
    r.from_aarch64 = FALSE;
    r.security     = security;
    r.regime       = regime;
    r.level        = TLBILevel_Any;
    r.attr         = attr;

    TLBI(r);
    if shareability != Shareability_NSH then Broadcast(shareability, r);
    return;
```

Library pseudocode for aarch32/functions/tlbi/AArch32.ITLBI_ASID

```
// AArch32.ITLBI_ASID()
// ====================
// Invalidate all instruction TLB stage 1 entries matching the indicated VMID
// (where regime supports) and ASID in the parameter Rt in the indicated translation
// regime with the indicated security state for all TLBs within the indicated shareability domain.
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.

AArch32.ITLBI_ASID(SecurityState security, Regime regime, bits(16) vmid,
                   Shareability shareability, TLBIMemAttr attr, bits(32) Rt)
    assert PSTATE.EL IN {EL3, EL2, EL1};

    TLBIRecord r;
    r.op           = TLBIOp_IASID;
    r.from_aarch64 = FALSE;
    r.security     = security;
    r.regime       = regime;
    r.vmid         = vmid;
    r.level        = TLBILevel_Any;
    r.attr         = attr;
    r.asid         = Zeros(8) : Rt<7:0>;

    TLBI(r);
    if shareability != Shareability_NSH then Broadcast(shareability, r);
    return;
```

Library pseudocode for aarch32/functions/tlbi/AArch32.ITLBI_VA

```
// AArch32.ITLBI_VA()
// ==================
// Invalidate by VA all stage 1 instruction TLB entries in the indicated shareability domain
// matching the indicated VMID and ASID (where regime supports VMID, ASID) in the indicated regime
// with the indicated security state.
// ASID, VA and related parameters are derived from Rt.
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.

AArch32.ITLBI_VA(SecurityState security, Regime regime, bits(16) vmid,
                 Shareability shareability, TLBILevel level,  TLBIMemAttr attr, bits(32) Rt)
    assert PSTATE.EL IN {EL3, EL2, EL1};

    TLBIRecord r;
    r.op           = TLBIOp_IVA;
    r.from_aarch64 = FALSE;
    r.security     = security;
    r.regime       = regime;
    r.vmid         = vmid;
    r.level        = level;
    r.attr         = attr;
    r.asid         = Zeros(8) : Rt<7:0>;
    r.address      = Zeros(32) : Rt<31:12> : Zeros(12);

    TLBI(r);
    if shareability != Shareability_NSH then Broadcast(shareability, r);
    return;
```

Library pseudocode for aarch32/functions/tlbi/AArch32.TLBI_ALL

```
// AArch32.TLBI_ALL()
// ==================
// Invalidate all entries for the indicated translation regime with the
// the indicated security state for all TLBs within the indicated shareability domain.
// Invalidation applies to all applicable stage 1 and stage 2 entries.

AArch32.TLBI_ALL(SecurityState security, Regime regime, Shareability shareability, TLBIMemAttr attr)
    assert PSTATE.EL IN {EL3, EL2};

    TLBIRecord r;
    r.op           = TLBIOp_ALL;
    r.from_aarch64 = FALSE;
    r.security     = security;
    r.regime       = regime;
    r.level        = TLBILevel_Any;
    r.attr         = attr;

    TLBI(r);
    if shareability != Shareability_NSH then Broadcast(shareability, r);
    return;
```

Library pseudocode for aarch32/functions/tlbi/AArch32.TLBI_ASID

```
// AArch32.TLBI_ASID()
// ===================
// Invalidate all stage 1 entries matching the indicated VMID (where regime supports)
// and ASID in the parameter Rt in the indicated translation regime with the
// indicated security state for all TLBs within the indicated shareability domain.
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.

AArch32.TLBI_ASID(SecurityState security, Regime regime, bits(16) vmid,
                  Shareability shareability, TLBIMemAttr attr, bits(32) Rt)
    assert PSTATE.EL IN {EL3, EL2, EL1};

    TLBIRecord r;
    r.op           = TLBIOp_ASID;
    r.from_aarch64 = FALSE;
    r.security     = security;
    r.regime       = regime;
    r.vmid         = vmid;
    r.level        = TLBILevel_Any;
    r.attr         = attr;
    r.asid         = Zeros(8) : Rt<7:0>;

    TLBI(r);
    if shareability != Shareability_NSH then Broadcast(shareability, r);
    return;
```

Library pseudocode for aarch32/functions/tlbi/AArch32.TLBI_IPAS2

```
// AArch32.TLBI_IPAS2()
// ====================
// Invalidate by IPA all stage 2 only TLB entries in the indicated shareability
// domain matching the indicated VMID in the indicated regime with the indicated security state.
// Note: stage 1 and stage 2 combined entries are not in the scope of this operation.
// IPA and related parameters of the are derived from Rt.

AArch32.TLBI_IPAS2(SecurityState security, Regime regime, bits(16) vmid,
                   Shareability shareability, TLBILevel level, TLBIMemAttr attr, bits(32) Rt)
    assert PSTATE.EL IN {EL3, EL2};
    assert security == SS_NonSecure;

    TLBIRecord r;
    r.op           = TLBIOp_IPAS2;
    r.from_aarch64 = FALSE;
    r.security     = security;
    r.regime       = regime;
    r.vmid         = vmid;
    r.level        = level;
    r.attr         = attr;
    r.address      = Zeros(24) : Rt<27:0> : Zeros(12);
    r.ipaspace     = PAS_NonSecure;

    TLBI(r);
    if shareability != Shareability_NSH then Broadcast(shareability, r);
    return;
```

Library pseudocode for aarch32/functions/tlbi/AArch32.TLBI_VA

```
// AArch32.TLBI_VA()
// =================
// Invalidate by VA all stage 1 TLB entries in the indicated shareability domain
// matching the indicated VMID and ASID (where regime supports VMID, ASID) in the indicated regime
// with the indicated security state.
// ASID, VA and related parameters are derived from Rt.
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.

AArch32.TLBI_VA(SecurityState security, Regime regime, bits(16) vmid,
                Shareability shareability, TLBILevel level,  TLBIMemAttr attr, bits(32) Rt)
    assert PSTATE.EL IN {EL3, EL2, EL1};

    TLBIRecord r;
    r.op           = TLBIOp_VA;
    r.from_aarch64 = FALSE;
    r.security     = security;
    r.regime       = regime;
    r.vmid         = vmid;
    r.level        = level;
    r.attr         = attr;
    r.asid         = Zeros(8) : Rt<7:0>;
    r.address      = Zeros(32) : Rt<31:12> : Zeros(12);

    TLBI(r);
    if shareability != Shareability_NSH then Broadcast(shareability, r);
    return;
```

Library pseudocode for aarch32/functions/tlbi/AArch32.TLBI_VAA

```
// AArch32.TLBI_VAA()
// ==================
// Invalidate by VA all stage 1 TLB entries in the indicated shareability domain
// matching the indicated VMID (where regime supports VMID) and all ASID in the indicated regime
// with the indicated security state.
// VA and related parameters are derived from Rt.
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.

AArch32.TLBI_VAA(SecurityState security, Regime regime, bits(16) vmid,
                 Shareability shareability, TLBILevel level, TLBIMemAttr attr,  bits(32) Rt)
    assert PSTATE.EL IN {EL3, EL2, EL1};

    TLBIRecord r;
    r.op           = TLBIOp_VAA;
    r.from_aarch64 = FALSE;
    r.security     = security;
    r.regime       = regime;
    r.vmid         = vmid;
    r.level        = level;
    r.attr         = attr;
    r.address      = Zeros(32) : Rt<31:12> : Zeros(12);

    TLBI(r);
    if shareability != Shareability_NSH then Broadcast(shareability, r);
    return;
```

Library pseudocode for aarch32/functions/tlbi/AArch32.TLBI_VMALL

```
// AArch32.TLBI_VMALL()
// ====================
// Invalidate all stage 1 entries for the indicated translation regime with the
// the indicated security state for all TLBs within the indicated shareability
// domain that match the indicated VMID (where applicable).
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.
// Note: stage 2 only entries are not in the scope of this operation.

AArch32.TLBI_VMALL(SecurityState security, Regime regime, bits(16) vmid,
                   Shareability shareability, TLBIMemAttr attr)
    assert PSTATE.EL IN {EL3, EL2, EL1};

    TLBIRecord r;
    r.op           = TLBIOp_VMALL;
    r.from_aarch64 = FALSE;
    r.security     = security;
    r.regime       = regime;
    r.level        = TLBILevel_Any;
    r.vmid         = vmid;
    r.attr         = attr;

    TLBI(r);
    if shareability != Shareability_NSH then Broadcast(shareability, r);
    return;
```

Library pseudocode for aarch32/functions/tlbi/AArch32.TLBI_VMALLS12

```
// AArch32.TLBI_VMALLS12()
// =======================
// Invalidate all stage 1 and stage 2 entries for the indicated translation
// regime with the indicated security state for all TLBs within the indicated
// shareability domain that match the indicated VMID.

AArch32.TLBI_VMALLS12(SecurityState security, Regime regime, bits(16) vmid,
                      Shareability shareability, TLBIMemAttr attr)
    assert PSTATE.EL IN {EL3, EL2};

    TLBIRecord r;
    r.op           = TLBIOp_VMALLS12;
    r.from_aarch64 = FALSE;
    r.security     = security;
    r.regime       = regime;
    r.level        = TLBILevel_Any;
    r.vmid         = vmid;
    r.attr         = attr;

    TLBI(r);
    if shareability != Shareability_NSH then Broadcast(shareability, r);
    return;
```

Library pseudocode for aarch32/functions/v6simd/Sat

```
// Sat()
// =====

bits(N) Sat(integer i, integer N, boolean unsigned)
    result = if unsigned then UnsignedSat(i, N) else SignedSat(i, N);
    return result;
```

Library pseudocode for aarch32/functions/v6simd/SignedSat

```
// SignedSat()
// ===========

bits(N) SignedSat(integer i, integer N)
    (result, -) = SignedSatQ(i, N);
    return result;
```

Library pseudocode for aarch32/functions/v6simd/UnsignedSat

```
// UnsignedSat()
// =============

bits(N) UnsignedSat(integer i, integer N)
    (result, -) = UnsignedSatQ(i, N);
    return result;
```

Library pseudocode for aarch32/ic/AArch32.IC

```
// AArch32.IC()
// ============
// Perform Instruction Cache Operation.

AArch32.IC(CacheOpScope opscope)
    regval = bits(32) UNKNOWN;
    AArch32.IC(regval, opscope);

// AArch32.IC()
// ============
// Perform Instruction Cache Operation.

AArch32.IC(bits(32) regval, CacheOpScope opscope)
    CacheRecord cache;

    cache.acctype   = AccessType_IC;
    cache.cachetype = CacheType_Instruction;
    cache.cacheop   = CacheOp_Invalidate;
    cache.opscope   = opscope;
    cache.security  = SecurityStateAtEL(PSTATE.EL);

    if opscope IN {CacheOpScope_ALLU, CacheOpScope_ALLUIS} then
        if opscope == CacheOpScope_ALLUIS || (opscope == CacheOpScope_ALLU && PSTATE.EL == EL1
                && EL2Enabled() && HCR.FB == '1') then
            cache.shareability = Shareability_ISH;
        else
            cache.shareability = Shareability_NSH;
        cache.regval = ZeroExtend(regval, 64);
        CACHE_OP(cache);
    else
        assert opscope == CacheOpScope_PoU;

        if EL2Enabled() then
            if PSTATE.EL IN {EL0, EL1} then
                cache.is_vmid_valid = TRUE;
                cache.vmid          = VMID[];
            else
                cache.is_vmid_valid = FALSE;
        else
            cache.is_vmid_valid = FALSE;

        if PSTATE.EL == EL0 then
            cache.is_asid_valid = TRUE;
            cache.asid          = ASID[];
        else
            cache.is_asid_valid = FALSE;

        need_translate = ICInstNeedsTranslation(opscope);

        cache.shareability = Shareability_NSH;
        cache.vaddress     = ZeroExtend(regval, 64);
        cache.translated   = need_translate;

        if !need_translate then
            cache.paddress = FullAddress UNKNOWN;
            CACHE_OP(cache);
            return;

        integer size    = 0;
        boolean aligned = TRUE;
        AccessDescriptor accdesc = CreateAccDescIC(cache);
        AddressDescriptor memaddrdesc = AArch32.TranslateAddress(regval, accdesc, aligned, size);
        if IsFault(memaddrdesc) then
            AArch32.Abort(regval, memaddrdesc.fault);

        cache.paddress = memaddrdesc.paddress;
        CACHE_OP(cache);
    return;
```

Library pseudocode for aarch32/predictionrestrict/AArch32.RestrictPrediction

```
// AArch32.RestrictPrediction()
// ============================
// Clear all predictions in the context.

AArch32.RestrictPrediction(bits(32) val, RestrictType restriction)

    ExecutionCntxt c;
    target_el    = val<25:24>;

    // If the target EL is not implemented or the instruction is executed at an
    // EL lower than the specified level, the instruction is treated as a NOP.
    if !HaveEL(target_el) || UInt(target_el) > UInt(PSTATE.EL) then EndOfInstruction();

    bit ns  = val<26>;
    bit nse = bit UNKNOWN;
    ss = TargetSecurityState(ns, nse);

    c.security  = ss;
    c.target_el = target_el;

    if EL2Enabled() then
        if PSTATE.EL IN {EL0, EL1} then
            c.is_vmid_valid = TRUE;
            c.all_vmid      = FALSE;
            c.vmid          = VMID[];

        elsif target_el IN {EL0, EL1} then
            c.is_vmid_valid = TRUE;
            c.all_vmid      = val<27> == '1';
            c.vmid          = ZeroExtend(val<23:16>, 16);      // Only valid if  val<27> == '0';
        else
            c.is_vmid_valid = FALSE;
    else
        c.is_vmid_valid = FALSE;

    if PSTATE.EL == EL0 then
        c.is_asid_valid = TRUE;
        c.all_asid      = FALSE;
        c.asid          = ASID[];

    elsif target_el == EL0 then
        c.is_asid_valid = TRUE;
        c.all_asid      = val<8> == '1';
        c.asid          = ZeroExtend(val<7:0>, 16);            // Only valid if  val<8> == '0';

    else
        c.is_asid_valid = FALSE;

    c.restriction = restriction;
    RESTRICT_PREDICTIONS(c);
```

Library pseudocode for aarch32/translation/attrs/AArch32.DefaultTEXDecode

```
// AArch32.DefaultTEXDecode()
// ==========================
// Apply short-descriptor format memory region attributes, without TEX remap

MemoryAttributes AArch32.DefaultTEXDecode(bits(3) TEX_in, bit C_in, bit B_in, bit s)
    MemoryAttributes memattrs;
    bits(3) TEX = TEX_in;
    bit C = C_in;
    bit B = B_in;

    // Reserved values map to allocated values
    if (TEX == '001' && C:B == '01') || (TEX == '010' && C:B != '00') || TEX == '011' then
        bits(5) texcb;
        (-, texcb) = ConstrainUnpredictableBits(Unpredictable_RESTEXCB, 5);
        TEX = texcb<4:2>;  C = texcb<1>;  B = texcb<0>;

    // Distinction between Inner Shareable and Outer Shareable is not supported in this format
    // A memory region is either Non-shareable or Outer Shareable
    case TEX:C:B of
        when '00000'
            // Device-nGnRnE
            memattrs.memtype      = MemType_Device;
            memattrs.device       = DeviceType_nGnRnE;
            memattrs.shareability = Shareability_OSH;
        when '00001', '01000'
            // Device-nGnRE
            memattrs.memtype      = MemType_Device;
            memattrs.device       = DeviceType_nGnRE;
            memattrs.shareability = Shareability_OSH;
        when '00010'
            // Write-through Read allocate
            memattrs.memtype      = MemType_Normal;
            memattrs.inner.attrs  = MemAttr_WT;
            memattrs.inner.hints  = MemHint_RA;
            memattrs.outer.attrs  = MemAttr_WT;
            memattrs.outer.hints  = MemHint_RA;
            memattrs.shareability = if s == '1' then Shareability_OSH else Shareability_NSH;
        when '00011'
            // Write-back Read allocate
            memattrs.memtype      = MemType_Normal;
            memattrs.inner.attrs  = MemAttr_WB;
            memattrs.inner.hints  = MemHint_RA;
            memattrs.outer.attrs  = MemAttr_WB;
            memattrs.outer.hints  = MemHint_RA;
            memattrs.shareability = if s == '1' then Shareability_OSH else Shareability_NSH;
        when '00100'
            // Non-cacheable
            memattrs.memtype      = MemType_Normal;
            memattrs.inner.attrs  = MemAttr_NC;
            memattrs.outer.attrs  = MemAttr_NC;
            memattrs.shareability = Shareability_OSH;
        when '00110'
            memattrs = MemoryAttributes IMPLEMENTATION_DEFINED;
        when '00111'
            // Write-back Read and Write allocate
            memattrs.memtype      = MemType_Normal;
            memattrs.inner.attrs  = MemAttr_WB;
            memattrs.inner.hints  = MemHint_RWA;
            memattrs.outer.attrs  = MemAttr_WB;
            memattrs.outer.hints  = MemHint_RWA;
            memattrs.shareability = if s == '1' then Shareability_OSH else Shareability_NSH;
        when '1xxxx'
            // Cacheable, TEX<1:0> = Outer attrs, {C,B} = Inner attrs
            memattrs.memtype = MemType_Normal;
            memattrs.inner   = DecodeSDFAttr(C:B);
            memattrs.outer   = DecodeSDFAttr(TEX<1:0>);

            if memattrs.inner.attrs == MemAttr_NC && memattrs.outer.attrs == MemAttr_NC then
                memattrs.shareability = Shareability_OSH;
            else
```

```
                memattrs.shareability = if s == '1' then Shareability_OSH else Shareability_NSH;
        otherwise
            // Reserved, handled above
            Unreachable();

    // The Transient hint is not supported in this format
    memattrs.inner.transient = FALSE;
    memattrs.outer.transient = FALSE;
    memattrs.tags            = MemTag_Untagged;

    if memattrs.inner.attrs == MemAttr_WB && memattrs.outer.attrs == MemAttr_WB then
        memattrs.xs = '0';
    else
        memattrs.xs = '1';

    return memattrs;
```

Library pseudocode for aarch32/translation/attrs/AArch32.MAIRAttr

```
// AArch32.MAIRAttr()
// ==================
// Retrieve the memory attribute encoding indexed in the given MAIR

bits(8) AArch32.MAIRAttr(integer index, MAIRType mair)
    assert (index < 8);
    bit_index = 8 * index;
    return mair<bit_index+7:bit_index>;
```

Library pseudocode for aarch32/translation/attrs/AArch32.RemappedTEXDecode

```
// AArch32.RemappedTEXDecode()
// ===========================
// Apply short-descriptor format memory region attributes, with TEX remap

MemoryAttributes AArch32.RemappedTEXDecode(Regime regime, bits(3) TEX, bit C, bit B, bit s)

    MemoryAttributes memattrs;
    PRRR_Type prrr;
    NMRR_Type nmrr;

    region = UInt(TEX<0>:C:B);         // TEX<2:1> are ignored in this mapping scheme
    if region == 6 then
        return MemoryAttributes IMPLEMENTATION_DEFINED;

    if regime == Regime_EL30 then
        prrr = PRRR_S;
        nmrr = NMRR_S;
    elsif HaveAArch32EL(EL3) then
        prrr = PRRR_NS;
        nmrr = NMRR_NS;
    else
        prrr = PRRR;
        nmrr = NMRR;

    base = 2 * region;
    attrfield = prrr<base+1:base>;

    if attrfield == '11' then      // Reserved, maps to allocated value
        (-, attrfield) = ConstrainUnpredictableBits(Unpredictable_RESPRRR, 2);

    case attrfield of
        when '00'                  // Device-nGnRnE
            memattrs.memtype      = MemType_Device;
            memattrs.device       = DeviceType_nGnRnE;
            memattrs.shareability = Shareability_OSH;
        when '01'                  // Device-nGnRE
            memattrs.memtype      = MemType_Device;
            memattrs.device       = DeviceType_nGnRE;
            memattrs.shareability = Shareability_OSH;
        when '10'
            NSn  = if s == '0' then prrr.NS0 else prrr.NS1;
            NOSm = prrr<region+24> AND NSn;
            IRn  = nmrr<base+1:base>;
            ORn  = nmrr<base+17:base+16>;

            memattrs.memtype = MemType_Normal;
            memattrs.inner   = DecodeSDFAttr(IRn);
            memattrs.outer   = DecodeSDFAttr(ORn);
            if memattrs.inner.attrs == MemAttr_NC && memattrs.outer.attrs == MemAttr_NC then
                memattrs.shareability = Shareability_OSH;
            else
                bits(2) sh = NSn:NOSm;
                memattrs.shareability = DecodeShareability(sh);
        when '11'
            Unreachable();

    // The Transient hint is not supported in this format
    memattrs.inner.transient = FALSE;
    memattrs.outer.transient = FALSE;
    memattrs.tags            = MemTag_Untagged;

    if memattrs.inner.attrs == MemAttr_WB && memattrs.outer.attrs == MemAttr_WB then
        memattrs.xs = '0';
    else
        memattrs.xs = '1';

    return memattrs;
```

Library pseudocode for aarch32/translation/debug/AArch32.CheckBreakpoint

```
// AArch32.CheckBreakpoint()
// =========================
// Called before executing the instruction of length "size" bytes at "vaddress" in an AArch32
// translation regime, when either debug exceptions are enabled, or halting debug is enabled
// and halting is allowed.

FaultRecord AArch32.CheckBreakpoint(FaultRecord fault_in, bits(32) vaddress,
                                    AccessDescriptor accdesc, integer size)
    assert ELUsingAArch32(S1TranslationRegime());
    assert size IN {2,4};

    FaultRecord fault = fault_in;
    match = FALSE;
    mismatch = FALSE;

    for i = 0 to NumBreakpointsImplemented() - 1
        (match_i, mismatch_i) = AArch32.BreakpointMatch(i, vaddress, accdesc, size);
        match = match || match_i;
        mismatch = mismatch || mismatch_i;

    if match && HaltOnBreakpointOrWatchpoint() then
        reason = DebugHalt_Breakpoint;
        Halt(reason);
    elsif (match || mismatch) then
        fault.statuscode = Fault_Debug;
        fault.debugmoe   = DebugException_Breakpoint;

    return fault;
```

Library pseudocode for aarch32/translation/debug/AArch32.CheckDebug

```
// AArch32.CheckDebug()
// ====================
// Called on each access to check for a debug exception or entry to Debug state.

FaultRecord AArch32.CheckDebug(bits(32) vaddress, AccessDescriptor accdesc, integer size)

    FaultRecord fault = NoFault(accdesc);

    boolean d_side = (IsDataAccess(accdesc.acctype) || accdesc.acctype == AccessType_DC);
    boolean i_side = (accdesc.acctype == AccessType_IFETCH);
    generate_exception = AArch32.GenerateDebugExceptions() && DBGDSCRext.MDBGen == '1';
    halt = HaltOnBreakpointOrWatchpoint();
    // Relative priority of Vector Catch and Breakpoint exceptions not defined in the architecture
    vector_catch_first = ConstrainUnpredictableBool(Unpredictable_BPVECTORCATCHPRI);

    if i_side && vector_catch_first && generate_exception then
        fault = AArch32.CheckVectorCatch(fault, vaddress, size);

    if fault.statuscode == Fault_None && (generate_exception || halt) then
        if d_side then
            fault = AArch32.CheckWatchpoint(fault, vaddress, accdesc, size);
        elsif i_side then
            fault = AArch32.CheckBreakpoint(fault, vaddress, accdesc, size);

    if fault.statuscode == Fault_None && i_side && !vector_catch_first && generate_exception then
        return AArch32.CheckVectorCatch(fault, vaddress, size);

    return fault;
```

Library pseudocode for aarch32/translation/debug/AArch32.CheckVectorCatch

```
// AArch32.CheckVectorCatch()
// ==========================
// Called before executing the instruction of length "size" bytes at "vaddress" in an AArch32
// translation regime, when debug exceptions are enabled.

FaultRecord AArch32.CheckVectorCatch(FaultRecord fault_in, bits(32) vaddress, integer size)
    assert ELUsingAArch32(S1TranslationRegime());

    FaultRecord fault = fault_in;
    match = AArch32.VCRMatch(vaddress);
    if size == 4 && !match && AArch32.VCRMatch(vaddress + 2) then
        match = ConstrainUnpredictableBool(Unpredictable_VCMATCHHALF);

    if match then
        fault.statuscode = Fault_Debug;
        fault.debugmoe   = DebugException_VectorCatch;

    return fault;
```

Library pseudocode for aarch32/translation/debug/AArch32.CheckWatchpoint

```
// AArch32.CheckWatchpoint()
// =========================
// Called before accessing the memory location of "size" bytes at "address",
// when either debug exceptions are enabled for the access, or halting debug
// is enabled and halting is allowed.

FaultRecord AArch32.CheckWatchpoint(FaultRecord fault_in, bits(32) vaddress,
                                    AccessDescriptor accdesc, integer size)
    assert ELUsingAArch32(S1TranslationRegime());
    FaultRecord fault = fault_in;

    if accdesc.acctype == AccessType_DC then
        if accdesc.cacheop != CacheOp_Invalidate then
            return fault;
        elsif !(boolean IMPLEMENTATION_DEFINED "DCIMVAC generates watchpoint") then
            return fault;
    elsif !IsDataAccess(accdesc.acctype) then
        return fault;

    match = FALSE;
    for i = 0 to NumWatchpointsImplemented() - 1
        if AArch32.WatchpointMatch(i, vaddress, size, accdesc) then
            match = TRUE;

    if match && HaltOnBreakpointOrWatchpoint() then
        reason = DebugHalt_Watchpoint;
        EDWAR = ZeroExtend(vaddress, 64);
        Halt(reason);
    elsif match then
        fault.statuscode = Fault_Debug;
        fault.debugmoe   = DebugException_Watchpoint;

    return fault;
```

Library pseudocode for aarch32/translation/faults/AArch32.IPAIsOutOfRange

```
// AArch32.IPAIsOutOfRange()
// =========================
// Check intermediate physical address bits not resolved by translation are ZERO

boolean AArch32.IPAIsOutOfRange(S2TTWParams walkparams, bits(40) ipa)
    // Input Address size
    iasize = AArch32.S2IASize(walkparams.t0sz);

    return iasize < 40 && !IsZero(ipa<39:iasize>);
```

Library pseudocode for aarch32/translation/faults/AArch32.S1HasAlignmentFault

```
// AArch32.S1HasAlignmentFault()
// =============================
// Returns whether stage 1 output fails alignment requirement on data accesses
// to Device memory

boolean AArch32.S1HasAlignmentFault(AccessDescriptor accdesc, boolean aligned,
                                    bit ntlsmd, MemoryAttributes memattrs)
    if accdesc.acctype == AccessType_IFETCH then
        return FALSE;
    elsif accdesc.a32lsmd && ntlsmd == '0' then
        return memattrs.memtype == MemType_Device &&  memattrs.device != DeviceType_GRE;
    elsif accdesc.acctype == AccessType_DCZero then
        return memattrs.memtype == MemType_Device;
    else
        return memattrs.memtype == MemType_Device && !aligned;
```

Library pseudocode for aarch32/translation/faults/AArch32.S1LDHasPermissionsFault

```
// AArch32.S1LDHasPermissionsFault()
// =================================
// Returns whether an access using stage 1 long-descriptor translation
// violates permissions of target memory

boolean AArch32.S1LDHasPermissionsFault(Regime regime, S1TTWParams walkparams, Permissions perms,
                                        MemType memtype, PASpace paspace, AccessDescriptor accdesc)
    bit  r,  w,  x;
    bit pr, pw;
    bit ur, uw;
    bit xn;
    if HasUnprivileged(regime) then
        // Apply leaf permissions
        case perms.ap<2:1> of
            when '00' (pr,pw,ur,uw) = ('1','1','0','0'); // R/W at PL1 only
            when '01' (pr,pw,ur,uw) = ('1','1','1','1'); // R/W at any PL
            when '10' (pr,pw,ur,uw) = ('1','0','0','0'); // RO at PL1 only
            when '11' (pr,pw,ur,uw) = ('1','0','1','0'); // RO at any PL

        // Apply hierarchical permissions
        case perms.ap_table of
            when '00' (pr,pw,ur,uw) = ( pr, pw, ur, uw); // No effect
            when '01' (pr,pw,ur,uw) = ( pr, pw,'0','0'); // Privileged access
            when '10' (pr,pw,ur,uw) = ( pr,'0', ur,'0'); // Read-only
            when '11' (pr,pw,ur,uw) = ( pr,'0','0','0'); // Read-only, privileged access

        xn   = perms.xn OR perms.xn_table;
        pxn  = perms.pxn OR perms.pxn_table;

        ux = ur AND NOT(xn OR (uw AND walkparams.wxn));
        px = pr AND NOT(xn OR pxn OR (pw AND walkparams.wxn) OR (uw AND walkparams.uwxn));

        if HavePANExt() && accdesc.pan then
            pan = PSTATE.PAN AND (ur OR uw);
            pr  = pr AND NOT(pan);
            pw  = pw AND NOT(pan);

        (r,w,x) = if accdesc.el == EL0 then (ur,uw,ux) else (pr,pw,px);

        // Prevent execution from Non-secure space by PE in Secure state if SIF is set
        if accdesc.ss == SS_Secure && paspace == PAS_NonSecure then
            x = x AND NOT(walkparams.sif);
    else
        // Apply leaf permissions
        case perms.ap<2> of
            when '0' (r,w) = ('1','1'); // No effect
            when '1' (r,w) = ('1','0'); // Read-only

        // Apply hierarchical permissions
        case perms.ap_table<1> of
            when '0' (r,w) = ( r , w ); // No effect
            when '1' (r,w) = ( r ,'0'); // Read-only

        xn = perms.xn OR perms.xn_table;
        x  = NOT(xn OR (w AND walkparams.wxn));

    if accdesc.acctype == AccessType_IFETCH then
        constraint = ConstrainUnpredictable(Unpredictable_INSTRDEVICE);
        if constraint == Constraint_FAULT && memtype == MemType_Device then
            return TRUE;
        else
            return x == '0';
    elsif accdesc.acctype IN {AccessType_IC, AccessType_DC} then
        return FALSE;
    elsif accdesc.write then
        return w == '0';
    else
        return r == '0';
```

Library pseudocode for aarch32/translation/faults/AArch32.S1SDHasPermissionsFault

```
// AArch32.S1SDHasPermissionsFault()
// =================================
// Returns whether an access using stage 1 short-descriptor translation
// violates permissions of target memory

boolean AArch32.S1SDHasPermissionsFault(Regime regime, Permissions perms_in, MemType memtype,
                                        PASpace paspace, AccessDescriptor accdesc)
    Permissions perms = perms_in;
    bit pr, pw;
    bit ur, uw;
    SCTLR_Type sctlr;
    if regime == Regime_EL30 then
        sctlr = SCTLR_S;
    elsif HaveAArch32EL(EL3) then
        sctlr = SCTLR_NS;
    else
        sctlr = SCTLR;

    if sctlr.AFE == '0' then
        // Map Reserved encoding '100'
        if perms.ap == '100' then
            perms.ap = bits(3) IMPLEMENTATION_DEFINED "Reserved short descriptor AP encoding";

        case perms.ap of
            when '000' (pr,pw,ur,uw) = ('0','0','0','0'); // No access
            when '001' (pr,pw,ur,uw) = ('1','1','0','0'); // R/W at PL1 only
            when '010' (pr,pw,ur,uw) = ('1','1','1','0'); // R/W at PL1, RO at PL0
            when '011' (pr,pw,ur,uw) = ('1','1','1','1'); // R/W at any PL
            //   '100' is reserved
            when '101' (pr,pw,ur,uw) = ('1','0','0','0'); // RO at PL1 only
            when '110' (pr,pw,ur,uw) = ('1','0','1','0'); // RO at any PL (deprecated)
            when '111' (pr,pw,ur,uw) = ('1','0','1','0'); // RO at any PL
    else // Simplified access permissions model
        case perms.ap<2:1> of
            when '00' (pr,pw,ur,uw) = ('1','1','0','0'); // R/W at PL1 only
            when '01' (pr,pw,ur,uw) = ('1','1','1','1'); // R/W at any PL
            when '10' (pr,pw,ur,uw) = ('1','0','0','0'); // RO at PL1 only
            when '11' (pr,pw,ur,uw) = ('1','0','1','0'); // RO at any PL

    ux = ur AND NOT(perms.xn OR (uw AND sctlr.WXN));
    px = pr AND NOT(perms.xn OR perms.pxn OR (pw AND sctlr.WXN) OR (uw AND sctlr.UWXN));

    if HavePANExt() && accdesc.pan then
        pan = PSTATE.PAN AND (ur OR uw);
        pr  = pr AND NOT(pan);
        pw  = pw AND NOT(pan);

    (r,w,x) = if accdesc.el == EL0 then (ur,uw,ux) else (pr,pw,px);

    // Prevent execution from Non-secure space by PE in Secure state if SIF is set
    if accdesc.ss == SS_Secure && paspace == PAS_NonSecure then
        x = x AND NOT(if ELUsingAArch32(EL3) then SCR.SIF else SCR_EL3.SIF);

    if accdesc.acctype == AccessType_IFETCH then
        if (memtype == MemType_Device &&
                ConstrainUnpredictable(Unpredictable_INSTRDEVICE) == Constraint_FAULT) then
            return TRUE;
        else
            return x == '0';
    elsif accdesc.acctype IN {AccessType_IC, AccessType_DC} then
        return FALSE;
    elsif accdesc.write then
        return w == '0';
    else
        return r == '0';
```

Library pseudocode for aarch32/translation/faults/AArch32.S2HasAlignmentFault

```
// AArch32.S2HasAlignmentFault()
// =============================
// Returns whether stage 2 output fails alignment requirement on data accesses
// to Device memory

boolean AArch32.S2HasAlignmentFault(AccessDescriptor accdesc, boolean aligned,
                                    MemoryAttributes memattrs)
    if accdesc.acctype == AccessType_IFETCH then
        return FALSE;
    elsif accdesc.acctype == AccessType_DCZero then
        return memattrs.memtype == MemType_Device;
    else
        return memattrs.memtype == MemType_Device && !aligned;
```

Library pseudocode for aarch32/translation/faults/AArch32.S2HasPermissionsFault

```
// AArch32.S2HasPermissionsFault()
// ===============================
// Returns whether stage 2 access violates permissions of target memory

boolean AArch32.S2HasPermissionsFault(S2TTWParams walkparams, Permissions perms, MemType memtype,
                                      AccessDescriptor accdesc)
    bit px;
    bit ux;
    r = perms.s2ap<0>;
    w = perms.s2ap<1>;
    bit x;
    if HaveExtendedExecuteNeverExt() then
        case perms.s2xn:perms.s2xnx of
            when '00'  (px, ux) = ( r , r );
            when '01'  (px, ux) = ('0', r );
            when '10'  (px, ux) = ('0','0');
            when '11'  (px, ux) = ( r ,'0');

        x = if accdesc.el == EL0 then ux else px;
    else
        x = r AND NOT(perms.s2xn);

    if accdesc.acctype == AccessType_TTW then
        return (walkparams.ptw == '1' && memtype == MemType_Device) || r == '0';

    elsif accdesc.acctype == AccessType_IFETCH then
        constraint = ConstrainUnpredictable(Unpredictable_INSTRDEVICE);
        return (constraint == Constraint_FAULT && memtype == MemType_Device) || x == '0';

    elsif accdesc.acctype IN {AccessType_IC, AccessType_DC} then
        return FALSE;

    elsif accdesc.write then
        return w == '0';

    else
        return r == '0';
```

Library pseudocode for aarch32/translation/faults/AArch32.S2InconsistentSL

```
// AArch32.S2InconsistentSL()
// ==========================
// Detect inconsistent configuration of stage 2 T0SZ and SL fields

boolean AArch32.S2InconsistentSL(S2TTWParams walkparams)
    startlevel  = AArch32.S2StartLevel(walkparams.sl0);
    levels      = FINAL_LEVEL - startlevel;
    granulebits = TGxGranuleBits(walkparams.tgx);
    stride      = granulebits - 3;

    // Input address size must at least be large enough to be resolved from the start level
    sl_min_iasize = (
        levels * stride // Bits resolved by table walk, except initial level
        + granulebits   // Bits directly mapped to output address
        + 1);           // At least 1 more bit to be decoded by initial level

    // Can accomodate 1 more stride in the level + concatenation of up to 2^4 tables
    sl_max_iasize = sl_min_iasize + (stride-1) + 4;
    // Configured Input Address size
    iasize        = AArch32.S2IASize(walkparams.t0sz);

    return iasize < sl_min_iasize || iasize > sl_max_iasize;
```

Library pseudocode for aarch32/translation/faults/AArch32.VAIsOutOfRange

```
// AArch32.VAIsOutOfRange()
// ========================
// Check virtual address bits not resolved by translation are identical
// and of accepted value

boolean AArch32.VAIsOutOfRange(Regime regime, S1TTWParams walkparams, bits(32) va)
    if regime == Regime_EL2 then
        // Input Address size
        iasize = AArch32.S1IASize(walkparams.t0sz);
        return walkparams.t0sz != '000' && !IsZero(va<31:iasize>);
    elsif walkparams.t1sz != '000' && walkparams.t0sz != '000' then
        // Lower range Input Address size
        lo_iasize = AArch32.S1IASize(walkparams.t0sz);
        // Upper range Input Address size
        up_iasize = AArch32.S1IASize(walkparams.t1sz);
        return !IsZero(va<31:lo_iasize>) && !IsOnes(va<31:up_iasize>);
    else
        return FALSE;
```

Library pseudocode for aarch32/translation/tlbcontext/AArch32.GetS1TLBContext

```
// AArch32.GetS1TLBContext()
// =========================
// Gather translation context for accesses with VA to match against TLB entries

TLBContext AArch32.GetS1TLBContext(Regime regime, SecurityState ss, bits(32) va)
    TLBContext tlbcontext;

    case regime of
        when Regime_EL2  tlbcontext = AArch32.TLBContextEL2(va);
        when Regime_EL10 tlbcontext = AArch32.TLBContextEL10(ss, va);
        when Regime_EL30 tlbcontext = AArch32.TLBContextEL30(va);

    tlbcontext.includes_s1 = TRUE;
    // The following may be amended for EL1&0 Regime if caching of stage 2 is successful
    tlbcontext.includes_s2 = FALSE;
    return tlbcontext;
```

Library pseudocode for aarch32/translation/tlbcontext/AArch32.GetS2TLBContext

```
// AArch32.GetS2TLBContext()
// =========================
// Gather translation context for accesses with IPA to match against TLB entries

TLBContext AArch32.GetS2TLBContext(FullAddress ipa)
    assert ipa.paspace == PAS_NonSecure;

    TLBContext tlbcontext;

    tlbcontext.ss          = SS_NonSecure;
    tlbcontext.regime      = Regime_EL10;
    tlbcontext.ipaspace    = ipa.paspace;
    tlbcontext.vmid        = ZeroExtend(VTTBR.VMID, 16);
    tlbcontext.tg          = TGx_4KB;
    tlbcontext.includes_s1 = FALSE;
    tlbcontext.includes_s2 = TRUE;
    tlbcontext.ia          = ZeroExtend(ipa.address, 64);
    tlbcontext.cnp         = if HaveCommonNotPrivateTransExt() then VTTBR.CnP else '0';

    return tlbcontext;
```

Library pseudocode for aarch32/translation/tlbcontext/AArch32.TLBContextEL10

```
// AArch32.TLBContextEL10()
// ========================
// Gather translation context for accesses under EL10 regime
// (PL10 when EL3 is A64) to match against TLB entries

TLBContext AArch32.TLBContextEL10(SecurityState ss, bits(32) va)
    TLBContext tlbcontext;
    TTBCR_Type ttbcr;
    TTBR0_Type ttbr0;
    TTBR1_Type ttbr1;
    CONTEXTIDR_Type contextidr;

    if HaveAArch32EL(EL3) then
        ttbcr      = TTBCR_NS;
        ttbr0      = TTBR0_NS;
        ttbr1      = TTBR1_NS;
        contextidr = CONTEXTIDR_NS;
    else
        ttbcr      = TTBCR;
        ttbr0      = TTBR0;
        ttbr1      = TTBR1;
        contextidr = CONTEXTIDR;

    tlbcontext.ss     = ss;
    tlbcontext.regime = Regime_EL10;

    if AArch32.EL2Enabled(ss) then
        tlbcontext.vmid = ZeroExtend(VTTBR.VMID, 16);

    if ttbcr.EAE == '1' then
        tlbcontext.asid = ZeroExtend(if ttbcr.A1 == '0' then ttbr0.ASID else ttbr1.ASID, 16);
    else
        tlbcontext.asid = ZeroExtend(contextidr.ASID, 16);

    tlbcontext.tg = TGx_4KB;
    tlbcontext.ia = ZeroExtend(va, 64);

    if HaveCommonNotPrivateTransExt() && ttbcr.EAE == '1' then
        if AArch32.GetVARange(va, ttbcr.T0SZ, ttbcr.T1SZ) == VARange_LOWER then
            tlbcontext.cnp = ttbr0.CnP;
        else
            tlbcontext.cnp = ttbr1.CnP;
    else
        tlbcontext.cnp = '0';

    return tlbcontext;
```

Library pseudocode for aarch32/translation/tlbcontext/AArch32.TLBContextEL2

```
// AArch32.TLBContextEL2()
// =======================
// Gather translation context for accesses under EL2 regime to match against TLB entries

TLBContext AArch32.TLBContextEL2(bits(32) va)
    TLBContext tlbcontext;

    tlbcontext.ss     = SS_NonSecure;
    tlbcontext.regime = Regime_EL2;
    tlbcontext.ia     = ZeroExtend(va, 64);
    tlbcontext.tg     = TGx_4KB;
    tlbcontext.cnp    = if HaveCommonNotPrivateTransExt() then HTTBR.CnP else '0';

    return tlbcontext;
```

Library pseudocode for aarch32/translation/tlbcontext/AArch32.TLBContextEL30

```
// AArch32.TLBContextEL30()
// ========================
// Gather translation context for accesses under EL30 regime
// (PL10 in Secure state and EL3 is A32) to match against TLB entries

TLBContext AArch32.TLBContextEL30(bits(32) va)
    TLBContext tlbcontext;

    tlbcontext.ss     = SS_Secure;
    tlbcontext.regime = Regime_EL30;

    if TTBCR_S.EAE == '1' then
        tlbcontext.asid = ZeroExtend(if TTBCR_S.A1 == '0' then TTBR0_S.ASID else TTBR1_S.ASID, 16);
    else
        tlbcontext.asid = ZeroExtend(CONTEXTIDR_S.ASID, 16);

    tlbcontext.tg = TGx_4KB;
    tlbcontext.ia = ZeroExtend(va, 64);

    if HaveCommonNotPrivateTransExt() && TTBCR_S.EAE == '1' then
        if AArch32.GetVARange(va, TTBCR_S.T0SZ, TTBCR_S.T1SZ) == VARange_LOWER then
            tlbcontext.cnp = TTBR0_S.CnP;
        else
            tlbcontext.cnp = TTBR1_S.CnP;
    else
        tlbcontext.cnp = '0';

    return tlbcontext;
```

Library pseudocode for aarch32/translation/translation/AArch32.EL2Enabled

```
// AArch32.EL2Enabled()
// ====================
// Returns whether EL2 is enabled for the given Security State

boolean AArch32.EL2Enabled(SecurityState ss)
    if ss == SS_Secure then
        if !(HaveEL(EL2) && HaveSecureEL2Ext()) then
            return FALSE;
        elsif HaveEL(EL3) then
            return SCR_EL3.EEL2 == '1';
        else
            return boolean IMPLEMENTATION_DEFINED "Secure-only implementation";
    else
        return HaveEL(EL2);
```

Library pseudocode for aarch32/translation/translation/AArch32.FullTranslate

```
// AArch32.FullTranslate()
// =======================
// Perform address translation as specified by VMSA-A32

AddressDescriptor AArch32.FullTranslate(bits(32) va, AccessDescriptor accdesc, boolean aligned)

    // Prepare fault fields in case a fault is detected
    FaultRecord fault = NoFault(accdesc);
    Regime regime = TranslationRegime(accdesc.el);

    // First Stage Translation
    AddressDescriptor ipa;
    if regime == Regime_EL2 || TTBCR.EAE == '1' then
        (fault, ipa) = AArch32.S1TranslateLD(fault, regime, va, aligned, accdesc);
    else
        (fault, ipa, -) = AArch32.S1TranslateSD(fault, regime, va, aligned, accdesc);

    if fault.statuscode != Fault_None then
        return CreateFaultyAddressDescriptor(ZeroExtend(va, 64), fault);

    if regime == Regime_EL10 && EL2Enabled() then
        ipa.vaddress = ZeroExtend(va, 64);
        AddressDescriptor pa;
        (fault, pa) = AArch32.S2Translate(fault, ipa, aligned, accdesc);

        if fault.statuscode != Fault_None then
            return CreateFaultyAddressDescriptor(ZeroExtend(va, 64), fault);
        else
            return pa;
    else
        return ipa;
```

Library pseudocode for aarch32/translation/translation/AArch32.OutputDomain

```
// AArch32.OutputDomain()
// ======================
// Determine the domain the translated output address

bits(2) AArch32.OutputDomain(Regime regime, bits(4) domain)
    bits(2) Dn;
    index = 2 * UInt(domain);
    if regime == Regime_EL30 then
        Dn = DACR_S<index+1:index>;
    elsif HaveAArch32EL(EL3) then
        Dn = DACR_NS<index+1:index>;
    else
        Dn = DACR<index+1:index>;

    if Dn == '10' then
        // Reserved value maps to an allocated value
        (-, Dn) = ConstrainUnpredictableBits(Unpredictable_RESDACR, 2);

    return Dn;
```

Library pseudocode for aarch32/translation/translation/AArch32.S1DisabledOutput

```
// AArch32.S1DisabledOutput()
// ==========================
// Flat map the VA to IPA/PA, depending on the regime, assigning default memory attributes

(FaultRecord, AddressDescriptor) AArch32.S1DisabledOutput(FaultRecord fault_in, Regime regime,
                                                          bits(32) va, boolean aligned,
                                                          AccessDescriptor accdesc)

    FaultRecord fault = fault_in;
    // No memory page is guarded when stage 1 address translation is disabled
    SetInGuardedPage(FALSE);

    MemoryAttributes memattrs;
    bit default_cacheable;
    if regime == Regime_EL10 && AArch32.EL2Enabled(accdesc.ss) then
        if ELStateUsingAArch32(EL2, accdesc.ss == SS_Secure) then
            default_cacheable = HCR.DC;
        else
            default_cacheable = HCR_EL2.DC;
    else
        default_cacheable = '0';

    if default_cacheable == '1' then
        // Use default cacheable settings
        memattrs.memtype      = MemType_Normal;
        memattrs.inner.attrs  = MemAttr_WB;
        memattrs.inner.hints  = MemHint_RWA;
        memattrs.outer.attrs  = MemAttr_WB;
        memattrs.outer.hints  = MemHint_RWA;
        memattrs.shareability = Shareability_NSH;
        if (!ELStateUsingAArch32(EL2, accdesc.ss == SS_Secure) &&
                HaveMTE2Ext() && HCR_EL2.DCT == '1') then
            memattrs.tags     = MemTag_AllocationTagged;
        else
            memattrs.tags     = MemTag_Untagged;
        memattrs.xs           = '0';
    elsif accdesc.acctype == AccessType_IFETCH then
        memattrs.memtype      = MemType_Normal;
        memattrs.shareability = Shareability_OSH;
        memattrs.tags         = MemTag_Untagged;
        if AArch32.S1ICacheEnabled(regime) then
            memattrs.inner.attrs = MemAttr_WT;
            memattrs.inner.hints = MemHint_RA;
            memattrs.outer.attrs = MemAttr_WT;
            memattrs.outer.hints = MemHint_RA;
        else
            memattrs.inner.attrs = MemAttr_NC;
            memattrs.outer.attrs = MemAttr_NC;
        memattrs.xs           = '1';
    else
        // Treat memory region as Device
        memattrs.memtype      = MemType_Device;
        memattrs.device       = DeviceType_nGnRnE;
        memattrs.shareability = Shareability_OSH;
        memattrs.tags         = MemTag_Untagged;
        memattrs.xs           = '1';

    bit ntlsmd;
    if HaveTrapLoadStoreMultipleDeviceExt() then
        case regime of
            when Regime_EL30 ntlsmd = SCTLR_S.nTLSMD;
            when Regime_EL2  ntlsmd = HSCTLR.nTLSMD;
            when Regime_EL10 ntlsmd = if HaveAArch32EL(EL3) then SCTLR_NS.nTLSMD else SCTLR.nTLSMD;
    else
        ntlsmd = '1';

    if AArch32.S1HasAlignmentFault(accdesc, aligned, ntlsmd, memattrs) then
        fault.statuscode  = Fault_Alignment;
        return (fault, AddressDescriptor UNKNOWN);
```

```
    FullAddress oa;
    oa.address = ZeroExtend(va, 56);
    oa.paspace = if accdesc.ss == SS_Secure then PAS_Secure else PAS_NonSecure;
    ipa = CreateAddressDescriptor(ZeroExtend(va, 64), oa, memattrs);

    return (fault, ipa);
```

Library pseudocode for aarch32/translation/translation/AArch32.S1Enabled

```
// AArch32.S1Enabled()
// ===================
// Returns whether stage 1 translation is enabled for the active translation regime

boolean AArch32.S1Enabled(Regime regime, SecurityState ss)
    if regime == Regime_EL2 then
        return HSCTLR.M == '1';
    elsif regime == Regime_EL30 then
        return SCTLR_S.M == '1';
    elsif !AArch32.EL2Enabled(ss) then
        return (if HaveAArch32EL(EL3) then SCTLR_NS.M else SCTLR.M) == '1';
    elsif ELStateUsingAArch32(EL2, ss == SS_Secure) then
        return HCR.<TGE,DC> == '00' && (if HaveAArch32EL(EL3) then SCTLR_NS.M else SCTLR.M) == '1';
    else
        return HCR_EL2.<TGE,DC> == '00' && SCTLR.M == '1';
```

Library pseudocode for aarch32/translation/translation/AArch32.S1TranslateLD

```
// AArch32.S1TranslateLD()
// =======================
// Perform a stage 1 translation using long-descriptor format mapping VA to IPA/PA
// depending on the regime

(FaultRecord, AddressDescriptor) AArch32.S1TranslateLD(FaultRecord fault_in, Regime regime,
                                                       bits(32) va, boolean aligned,
                                                       AccessDescriptor accdesc)
    FaultRecord fault = fault_in;

    if !AArch32.S1Enabled(regime, accdesc.ss) then
        return AArch32.S1DisabledOutput(fault, regime, va, aligned, accdesc);

    walkparams = AArch32.GetS1TTWParams(regime, va);

    if AArch32.VAIsOutOfRange(regime, walkparams, va) then
        fault.level      = 1;
        fault.statuscode = Fault_Translation;
        return (fault, AddressDescriptor UNKNOWN);

    TTWState walkstate;
    (fault, walkstate) = AArch32.S1WalkLD(fault, regime, walkparams, accdesc, va);

    if fault.statuscode != Fault_None then
        return (fault, AddressDescriptor UNKNOWN);

    SetInGuardedPage(FALSE);  // AArch32-VMSA does not guard any pages

    if AArch32.S1HasAlignmentFault(accdesc, aligned, walkparams.ntlsmd, walkstate.memattrs) then
        fault.statuscode = Fault_Alignment;
    elsif AArch32.S1LDHasPermissionsFault(regime, walkparams,
                                          walkstate.permissions,
                                          walkstate.memattrs.memtype,
                                          walkstate.baseaddress.paspace,
                                          accdesc) then
        fault.statuscode = Fault_Permission;

    if fault.statuscode != Fault_None then
        return (fault, AddressDescriptor UNKNOWN);

    MemoryAttributes memattrs;
    if ((accdesc.acctype == AccessType_IFETCH &&
            (walkstate.memattrs.memtype == MemType_Device || !AArch32.S1ICacheEnabled(regime))) ||
        (accdesc.acctype != AccessType_IFETCH &&
             walkstate.memattrs.memtype == MemType_Normal && !AArch32.S1DCacheEnabled(regime))) then
        // Treat memory attributes as Normal Non-Cacheable
        memattrs = NormalNCMemAttr();
        memattrs.xs = walkstate.memattrs.xs;
    else
        memattrs = walkstate.memattrs;

    // Shareability value of stage 1 translation subject to stage 2 is IMPLEMENTATION DEFINED
    // to be either effective value or descriptor value
    if (regime == Regime_EL10 && AArch32.EL2Enabled(accdesc.ss) &&
        (if ELStateUsingAArch32(EL2, accdesc.ss==SS_Secure) then HCR.VM else HCR_EL2.VM) == '1' &&
        !(boolean IMPLEMENTATION_DEFINED "Apply effective shareability at stage 1")) then
        memattrs.shareability = walkstate.memattrs.shareability;
    else
        memattrs.shareability = EffectiveShareability(memattrs);

    // Output Address
    oa = StageOA(ZeroExtend(va, 64), walkparams.d128, walkparams.tgx, walkstate);
    ipa = CreateAddressDescriptor(ZeroExtend(va, 64), oa, memattrs);

    return (fault, ipa);
```

Library pseudocode for aarch32/translation/translation/AArch32.S1TranslateSD

```
// AArch32.S1TranslateSD()
// =======================
// Perform a stage 1 translation using short-descriptor format mapping VA to IPA/PA
// depending on the regime

(FaultRecord, AddressDescriptor, SDFType) AArch32.S1TranslateSD(FaultRecord fault_in, Regime regime,
                                                                bits(32) va, boolean aligned,
                                                                AccessDescriptor accdesc)

    FaultRecord fault = fault_in;

    if !AArch32.S1Enabled(regime, accdesc.ss) then
        AddressDescriptor ipa;
        (fault, ipa) = AArch32.S1DisabledOutput(fault, regime, va, aligned, accdesc);
        return (fault, ipa, SDFType UNKNOWN);

    TTWState walkstate;
    (fault, walkstate) = AArch32.S1WalkSD(fault, regime, accdesc, va);

    if fault.statuscode != Fault_None then
        return (fault, AddressDescriptor UNKNOWN, SDFType UNKNOWN);

    domain = AArch32.OutputDomain(regime, walkstate.domain);
    SetInGuardedPage(FALSE);  // AArch32-VMSA does not guard any pages

    bit ntlsmd;
    if HaveTrapLoadStoreMultipleDeviceExt() then
        case regime of
            when Regime_EL30 ntlsmd = SCTLR_S.nTLSMD;
            when Regime_EL10 ntlsmd = if HaveAArch32EL(EL3) then SCTLR_NS.nTLSMD else SCTLR.nTLSMD;
    else
        ntlsmd = '1';

    if AArch32.S1HasAlignmentFault(accdesc, aligned, ntlsmd, walkstate.memattrs) then
        fault.statuscode = Fault_Alignment;
    elsif (!(accdesc.acctype IN {AccessType_IC, AccessType_DC}) &&
            domain == Domain_NoAccess) then
        fault.statuscode = Fault_Domain;
    elsif domain == Domain_Client then
        if AArch32.S1SDHasPermissionsFault(regime, walkstate.permissions,
                                           walkstate.memattrs.memtype,
                                           walkstate.baseaddress.paspace,
                                           accdesc) then
            fault.statuscode = Fault_Permission;

    if fault.statuscode != Fault_None then
        fault.domain = walkstate.domain;
        return (fault, AddressDescriptor UNKNOWN, walkstate.sdftype);

    MemoryAttributes memattrs;
    if ((accdesc.acctype == AccessType_IFETCH &&
            (walkstate.memattrs.memtype == MemType_Device || !AArch32.S1ICacheEnabled(regime))) ||
        (accdesc.acctype != AccessType_IFETCH &&
             walkstate.memattrs.memtype == MemType_Normal && !AArch32.S1DCacheEnabled(regime))) then
        // Treat memory attributes as Normal Non-Cacheable
        memattrs = NormalNCMemAttr();
        memattrs.xs = walkstate.memattrs.xs;
    else
        memattrs = walkstate.memattrs;

    // Shareability value of stage 1 translation subject to stage 2 is IMPLEMENTATION DEFINED
    // to be either effective value or descriptor value
    if (regime == Regime_EL10 && AArch32.EL2Enabled(accdesc.ss) &&
        (if ELStateUsingAArch32(EL2, accdesc.ss==SS_Secure) then HCR.VM else HCR_EL2.VM) == '1' &&
        !(boolean IMPLEMENTATION_DEFINED "Apply effective shareability at stage 1")) then
        memattrs.shareability = walkstate.memattrs.shareability;
    else
        memattrs.shareability = EffectiveShareability(memattrs);

    // Output Address
```

```
    oa = AArch32.SDStageOA(walkstate.baseaddress, va, walkstate.sdftype);
    ipa = CreateAddressDescriptor(ZeroExtend(va, 64), oa, memattrs);

    return (fault, ipa, walkstate.sdftype);
```

Library pseudocode for aarch32/translation/translation/AArch32.S2Translate

```
// AArch32.S2Translate()
// =====================
// Perform a stage 2 translation mapping an IPA to a PA

(FaultRecord, AddressDescriptor) AArch32.S2Translate(FaultRecord fault_in, AddressDescriptor ipa,
                                                     boolean aligned, AccessDescriptor accdesc)

    FaultRecord fault = fault_in;
    assert IsZero(ipa.paddress.address<55:40>);

    if !ELStateUsingAArch32(EL2, accdesc.ss == SS_Secure) then
        s1aarch64 = FALSE;
        return AArch64.S2Translate(fault, ipa, s1aarch64, aligned, accdesc);

    // Prepare fault fields in case a fault is detected
    fault.statuscode  = Fault_None;
    fault.secondstage = TRUE;
    fault.s2fs1walk   = accdesc.acctype == AccessType_TTW;
    fault.ipaddress   = ipa.paddress;

    walkparams = AArch32.GetS2TTWParams();

    if walkparams.vm == '0' then
        // Stage 2 is disabled
        return (fault, ipa);

    if AArch32.IPAIsOutOfRange(walkparams, ipa.paddress.address<39:0>) then
        fault.statuscode = Fault_Translation;
        fault.level      = 1;
        return (fault, AddressDescriptor UNKNOWN);

    TTWState walkstate;
    (fault, walkstate) = AArch32.S2Walk(fault, walkparams, accdesc, ipa);

    if fault.statuscode != Fault_None then
        return (fault, AddressDescriptor UNKNOWN);

    if AArch32.S2HasAlignmentFault(accdesc, aligned, walkstate.memattrs) then
        fault.statuscode = Fault_Alignment;
    elsif AArch32.S2HasPermissionsFault(walkparams,
                                        walkstate.permissions,
                                        walkstate.memattrs.memtype,
                                        accdesc) then
        fault.statuscode = Fault_Permission;
    MemoryAttributes s2_memattrs;
    if ((accdesc.acctype == AccessType_TTW &&
             walkstate.memattrs.memtype == MemType_Device) ||
        (accdesc.acctype == AccessType_IFETCH &&
            (walkstate.memattrs.memtype == MemType_Device || HCR2.ID == '1')) ||
        (accdesc.acctype != AccessType_IFETCH &&
             walkstate.memattrs.memtype == MemType_Normal && HCR2.CD == '1')) then
        // Treat memory attributes as Normal Non-Cacheable
        s2_memattrs = NormalNCMemAttr();
        s2_memattrs.xs = walkstate.memattrs.xs;
    else
        s2_memattrs = walkstate.memattrs;

    s2aarch64 = FALSE;
    memattrs = S2CombineS1MemAttrs(ipa.memattrs, s2_memattrs, s2aarch64);
    ipa_64 = ZeroExtend(ipa.paddress.address<39:0>, 64);
    // Output Address
    oa = StageOA(ipa_64, walkparams.d128, walkparams.tgx, walkstate);
    pa = CreateAddressDescriptor(ipa.vaddress, oa, memattrs);

    return (fault, pa);
```

Library pseudocode for aarch32/translation/translation/AArch32.SDStageOA

```
// AArch32.SDStageOA()
// ===================
// Given the final walk state of a short-descriptor translation walk,
// map the untranslated input address bits to the base output address

FullAddress AArch32.SDStageOA(FullAddress baseaddress, bits(32) va, SDFType sdftype)
    integer tsize;
    case sdftype of
        when SDFType_SmallPage      tsize = 12;
        when SDFType_LargePage      tsize = 16;
        when SDFType_Section        tsize = 20;
        when SDFType_Supersection   tsize = 24;

    // Output Address
    FullAddress oa;
    oa.address = baseaddress.address<55:tsize>:va<tsize-1:0>;
    oa.paspace = baseaddress.paspace;
    return oa;
```

Library pseudocode for aarch32/translation/translation/AArch32.TranslateAddress

```
// AArch32.TranslateAddress()
// ==========================
// Main entry point for translating an address

AddressDescriptor AArch32.TranslateAddress(bits(32) va, AccessDescriptor accdesc,
                                           boolean aligned, integer size)

    Regime regime = TranslationRegime(PSTATE.EL);
    if !RegimeUsingAArch32(regime) then
        return AArch64.TranslateAddress(ZeroExtend(va, 64), accdesc, aligned, size);

    AddressDescriptor result = AArch32.FullTranslate(va, accdesc, aligned);

    if !IsFault(result) then
        result.fault = AArch32.CheckDebug(va, accdesc, size);

    // Update virtual address for abort functions
    result.vaddress = ZeroExtend(va, 64);

    return result;
```

Library pseudocode for aarch32/translation/walk/AArch32.DecodeDescriptorTypeLD

```
// AArch32.DecodeDescriptorTypeLD()
// ================================
// Determine whether the long-descriptor is a page, block or table

DescriptorType AArch32.DecodeDescriptorTypeLD(bits(64) descriptor, integer level)
    if descriptor<1:0> == '11' && level == FINAL_LEVEL then
        return DescriptorType_Leaf;
    elsif descriptor<1:0> == '11' then
        return DescriptorType_Table;
    elsif descriptor<1:0> == '01' && level != FINAL_LEVEL then
        return DescriptorType_Leaf;
    else
        return DescriptorType_Invalid;
```

Library pseudocode for aarch32/translation/walk/AArch32.DecodeDescriptorTypeSD

```
// AArch32.DecodeDescriptorTypeSD()
// ================================
// Determine the type of the short-descriptor

SDFType AArch32.DecodeDescriptorTypeSD(bits(32) descriptor, integer level)
    if level == 1 && descriptor<1:0> == '01' then
        return SDFType_Table;
    elsif level == 1 && descriptor<18,1> == '01' then
        return SDFType_Section;
    elsif level == 1 && descriptor<18,1> == '11' then
        return SDFType_Supersection;
    elsif level == 2 && descriptor<1:0> == '01' then
        return SDFType_LargePage;
    elsif level == 2 && descriptor<1:0> IN {'1x'} then
        return SDFType_SmallPage;
    else
        return SDFType_Invalid;
```

Library pseudocode for aarch32/translation/walk/AArch32.S1IASize

```
// AArch32.S1IASize()
// ==================
// Retrieve the number of bits containing the input address for stage 1 translation

integer AArch32.S1IASize(bits(3) txsz)
    return 32 - UInt(txsz);
```

Library pseudocode for aarch32/translation/walk/AArch32.S1WalkLD

```
// AArch32.S1WalkLD()
// ==================
// Traverse stage 1 translation tables in long format to obtain the final descriptor

(FaultRecord, TTWState) AArch32.S1WalkLD(FaultRecord fault_in, Regime regime,
                                         S1TTWParams walkparams, AccessDescriptor accdesc,
                                         bits(32) va)
    FaultRecord fault = fault_in;
    bits(3) txsz;
    bits(64) ttbr;
    bit epd;
    VARange varange;
    if regime == Regime_EL2 then
        ttbr = HTTBR;
        txsz = walkparams.t0sz;
        varange = VARange_LOWER;
    else
        varange = AArch32.GetVARange(va, walkparams.t0sz, walkparams.t1sz);
        bits(64) ttbr0;
        bits(64) ttbr1;
        TTBCR_Type ttbcr;
        if regime == Regime_EL30 then
            ttbcr = TTBCR_S;
            ttbr0 = TTBR0_S;
            ttbr1 = TTBR1_S;
        elsif HaveAArch32EL(EL3) then
            ttbcr = TTBCR_NS;
            ttbr0 = TTBR0_NS;
            ttbr1 = TTBR1_NS;
        else
            ttbcr = TTBCR;
            ttbr0 = TTBR0;
            ttbr1 = TTBR1;

        assert ttbcr.EAE == '1';
        if varange == VARange_LOWER then
            txsz = walkparams.t0sz;
            ttbr = ttbr0;
            epd  = ttbcr.EPD0;
        else
            txsz = walkparams.t1sz;
            ttbr = ttbr1;
            epd  = ttbcr.EPD1;

    if regime != Regime_EL2 && epd == '1' then
        fault.level      = 1;
        fault.statuscode = Fault_Translation;
        return (fault, TTWState UNKNOWN);

    // Input Address size
    iasize      = AArch32.S1IASize(txsz);
    granulebits = TGxGranuleBits(walkparams.tgx);
    stride      = granulebits - 3;
    startlevel  = FINAL_LEVEL - (((iasize-1) - granulebits) DIV stride);
    levels      = FINAL_LEVEL - startlevel;

    if !IsZero(ttbr<47:40>) then
        fault.statuscode = Fault_AddressSize;
        fault.level      = 0;
        return (fault, TTWState UNKNOWN);

    FullAddress baseaddress;
    baselsb = (iasize - (levels*stride + granulebits)) + 3;
    baseaddress.paspace = if accdesc.ss == SS_Secure then PAS_Secure else PAS_NonSecure;
    baseaddress.address = ZeroExtend(ttbr<39:baselsb>:Zeros(baselsb), 56);

    TTWState walkstate;
    walkstate.baseaddress = baseaddress;
    walkstate.level       = startlevel;
    walkstate.istable     = TRUE;
```

```
    // In regimes that support global and non-global translations, translation
    // table entries from lookup levels other than the final level of lookup
    // are treated as being non-global
    walkstate.nG          = if HasUnprivileged(regime) then '1' else '0';
    walkstate.memattrs    = WalkMemAttrs(walkparams.sh, walkparams.irgn, walkparams.orgn);
    walkstate.permissions.ap_table  = '00';
    walkstate.permissions.xn_table  = '0';
    walkstate.permissions.pxn_table = '0';

    indexmsb = iasize - 1;
    bits(64) descriptor;
    AddressDescriptor walkaddress;

    walkaddress.vaddress = ZeroExtend(va, 64);

    if !AArch32.S1DCacheEnabled(regime) then
        walkaddress.memattrs = NormalNCMemAttr();
        walkaddress.memattrs.xs = walkstate.memattrs.xs;
    else
        walkaddress.memattrs = walkstate.memattrs;

    // Shareability value of stage 1 translation subject to stage 2 is IMPLEMENTATION DEFINED
    // to be either effective value or descriptor value
    if (regime == Regime_EL10 && AArch32.EL2Enabled(accdesc.ss) &&
        (if ELStateUsingAArch32(EL2, accdesc.ss==SS_Secure) then HCR.VM else HCR_EL2.VM) == '1' &&
        !(boolean IMPLEMENTATION_DEFINED "Apply effective shareability at stage 1")) then
        walkaddress.memattrs.shareability = walkstate.memattrs.shareability;
    else
        walkaddress.memattrs.shareability = EffectiveShareability(walkaddress.memattrs);

    integer indexlsb;
    DescriptorType desctype;
    repeat
        fault.level = walkstate.level;
        indexlsb = (FINAL_LEVEL - walkstate.level)*stride + granulebits;
        bits(40) index = ZeroExtend(va<indexmsb:indexlsb>:'000', 40);

        walkaddress.paddress.address = walkstate.baseaddress.address OR ZeroExtend(index, 56);
        walkaddress.paddress.paspace = walkstate.baseaddress.paspace;

        boolean toplevel = walkstate.level == startlevel;
        AccessDescriptor walkaccess = CreateAccDescS1TTW(toplevel, varange, accdesc);
        // If there are two stages of translation, then the first stage table walk addresses
        // are themselves subject to translation
        if regime == Regime_EL10 && AArch32.EL2Enabled(accdesc.ss) then
            s2aligned = TRUE;
            (s2fault, s2walkaddress) = AArch32.S2Translate(fault, walkaddress, s2aligned,
                                                           walkaccess);
            // Check for a fault on the stage 2 walk
            if s2fault.statuscode != Fault_None then
                return (s2fault, TTWState UNKNOWN);

            (fault, descriptor) = FetchDescriptor(walkparams.ee, s2walkaddress, walkaccess,
                                                  fault, 64);
        else
            (fault, descriptor) = FetchDescriptor(walkparams.ee, walkaddress, walkaccess,
                                                  fault, 64);

        if fault.statuscode != Fault_None then
            return (fault, TTWState UNKNOWN);

        desctype = AArch32.DecodeDescriptorTypeLD(descriptor, walkstate.level);

        case desctype of
            when DescriptorType_Table
                if !IsZero(descriptor<47:40>) then
                    fault.statuscode = Fault_AddressSize;
                    return (fault, TTWState UNKNOWN);

                walkstate.baseaddress.address = ZeroExtend(descriptor<39:12>:Zeros(12), 56);
```

```
            if walkstate.baseaddress.paspace == PAS_Secure && descriptor<63> == '1' then
                walkstate.baseaddress.paspace = PAS_NonSecure;

            if walkparams.hpd == '0' then
                walkstate.permissions.xn_table  = (walkstate.permissions.xn_table  OR
                                                    descriptor<60>);
                walkstate.permissions.ap_table  = (walkstate.permissions.ap_table  OR
                                                    descriptor<62:61>);
                walkstate.permissions.pxn_table = (walkstate.permissions.pxn_table OR
                                                    descriptor<59>);

            walkstate.level = walkstate.level + 1;
            indexmsb = indexlsb - 1;

        when DescriptorType_Invalid
            fault.statuscode = Fault_Translation;
            return (fault, TTWState UNKNOWN);

        when DescriptorType_Leaf
            walkstate.istable = FALSE;

until desctype == DescriptorType_Leaf;

// Check the output address is inside the supported range
if !IsZero(descriptor<47:40>) then
    fault.statuscode = Fault_AddressSize;
    return (fault, TTWState UNKNOWN);

// Check the access flag
if descriptor<10> == '0' then
    fault.statuscode = Fault_AccessFlag;
    return (fault, TTWState UNKNOWN);

walkstate.permissions.xn  = descriptor<54>;
walkstate.permissions.pxn = descriptor<53>;
walkstate.permissions.ap  = descriptor<7:6>:'1';
walkstate.contiguous      = descriptor<52>;
if regime == Regime_EL2 then
    // All EL2 regime accesses are treated as Global
    walkstate.nG = '0';
elsif accdesc.ss == SS_Secure && walkstate.baseaddress.paspace == PAS_NonSecure then
    // When a PE is using the Long-descriptor translation table format,
    // and is in Secure state, a translation must be treated as non-global,
    // regardless of the value of the nG bit,
    // if NSTable is set to 1 at any level of the translation table walk.
    walkstate.nG = '1';
else
    walkstate.nG = descriptor<11>;

walkstate.baseaddress.address = ZeroExtend(descriptor<39:indexlsb>:Zeros(indexlsb), 56);
if walkstate.baseaddress.paspace == PAS_Secure && descriptor<5> == '1' then
    walkstate.baseaddress.paspace = PAS_NonSecure;

memattr = descriptor<4:2>;
sh      = descriptor<9:8>;
attr    = AArch32.MAIRAttr(UInt(memattr), walkparams.mair);
s1aarch64 = FALSE;
walkstate.memattrs = S1DecodeMemAttrs(attr, sh, s1aarch64, walkparams);

return (fault, walkstate);
```

Library pseudocode for aarch32/translation/walk/AArch32.S1WalkSD

```
// AArch32.S1WalkSD()
// ==================
// Traverse stage 1 translation tables in short format to obtain the final descriptor

(FaultRecord, TTWState) AArch32.S1WalkSD(FaultRecord fault_in, Regime regime,
                                         AccessDescriptor accdesc, bits(32) va)
    FaultRecord fault = fault_in;
    SCTLR_Type sctlr;
    TTBCR_Type ttbcr;
    TTBR0_Type ttbr0;
    TTBR1_Type ttbr1;
    // Determine correct translation control registers to use.
    if regime == Regime_EL30 then
        sctlr = SCTLR_S;
        ttbcr = TTBCR_S;
        ttbr0 = TTBR0_S;
        ttbr1 = TTBR1_S;
    elsif HaveAArch32EL(EL3) then
        sctlr = SCTLR_NS;
        ttbcr = TTBCR_NS;
        ttbr0 = TTBR0_NS;
        ttbr1 = TTBR1_NS;
    else
        sctlr = SCTLR;
        ttbcr = TTBCR;
        ttbr0 = TTBR0;
        ttbr1 = TTBR1;

    assert ttbcr.EAE == '0';
    ee  = sctlr.EE;
    afe = sctlr.AFE;
    tre = sctlr.TRE;
    n = UInt(ttbcr.N);
    bits(32) ttb;
    bits(1) pd;
    bits(2) irgn;
    bits(2) rgn;
    bits(1) s;
    bits(1) nos;
    VARange varange;
    if n == 0 || IsZero(va<31:(32-n)>) then
        ttb  = ttbr0.TTB0:Zeros(7);
        pd   = ttbcr.PD0;
        irgn = ttbr0.IRGN;
        rgn  = ttbr0.RGN;
        s    = ttbr0.S;
        nos  = ttbr0.NOS;
        varange = VARange_LOWER;
    else
        n    = 0;  // TTBR1 translation always treats N as 0
        ttb  = ttbr1.TTB1:Zeros(7);
        pd   = ttbcr.PD1;
        irgn = ttbr1.IRGN;
        rgn  = ttbr1.RGN;
        s    = ttbr1.S;
        nos  = ttbr1.NOS;
        varange = VARange_UPPER;

    // Check if Translation table walk disabled for translations with this Base register.
    if pd == '1' then
        fault.level      = 1;
        fault.statuscode = Fault_Translation;
        return (fault, TTWState UNKNOWN);

    FullAddress baseaddress;
    baseaddress.paspace = if accdesc.ss == SS_Secure then PAS_Secure else PAS_NonSecure;
    baseaddress.address = ZeroExtend(ttb<31:14-n>:Zeros(14-n), 56);

    constant integer startlevel = 1;
    TTWState walkstate;
```

```
    walkstate.baseaddress = baseaddress;
    // In regimes that support global and non-global translations, translation
    // table entries from lookup levels other than the final level of lookup
    // are treated as being non-global. Translations in Short-Descriptor Format
    // always support global & non-global translations.
    walkstate.nG          = '1';
    walkstate.memattrs    = WalkMemAttrs(s:nos, irgn, rgn);
    walkstate.level       = startlevel;
    walkstate.istable     = TRUE;

    bits(4) domain;
    bits(32) descriptor;
    AddressDescriptor walkaddress;

    walkaddress.vaddress = ZeroExtend(va, 64);

    if !AArch32.S1DCacheEnabled(regime) then
        walkaddress.memattrs = NormalNCMemAttr();
        walkaddress.memattrs.xs = walkstate.memattrs.xs;
    else
        walkaddress.memattrs = walkstate.memattrs;

    // Shareability value of stage 1 translation subject to stage 2 is IMPLEMENTATION DEFINED
    // to be either effective value or descriptor value
    if (regime == Regime_EL10 && AArch32.EL2Enabled(accdesc.ss) &&
        (if ELStateUsingAArch32(EL2, accdesc.ss==SS_Secure) then HCR.VM else HCR_EL2.VM) == '1' &&
        !(boolean IMPLEMENTATION_DEFINED "Apply effective shareability at stage 1")) then
        walkaddress.memattrs.shareability = walkstate.memattrs.shareability;
    else
        walkaddress.memattrs.shareability = EffectiveShareability(walkaddress.memattrs);

    bit nG;
    bit ns;
    bit pxn;
    bits(3) ap;
    bits(3) tex;
    bit c;
    bit b;
    bit xn;
    repeat
        fault.level = walkstate.level;

        bits(32) index;
        if walkstate.level == 1 then
            index = ZeroExtend(va<31-n:20>:'00', 32);
        else
            index = ZeroExtend(va<19:12>:'00', 32);

        walkaddress.paddress.address = walkstate.baseaddress.address OR ZeroExtend(index,
                                                                                    56);
        walkaddress.paddress.paspace = walkstate.baseaddress.paspace;

        boolean toplevel = walkstate.level == startlevel;
        AccessDescriptor walkaccess = CreateAccDescS1TTW(toplevel, varange, accdesc);
        if regime == Regime_EL10 && AArch32.EL2Enabled(accdesc.ss) then
            s2aligned = TRUE;
            (s2fault, s2walkaddress) = AArch32.S2Translate(fault, walkaddress, s2aligned,
                                                           walkaccess);

            if s2fault.statuscode != Fault_None then
                return (s2fault, TTWState UNKNOWN);

            (fault, descriptor) = FetchDescriptor(ee, s2walkaddress, walkaccess, fault, 32);
        else
            (fault, descriptor) = FetchDescriptor(ee, walkaddress, walkaccess, fault, 32);

        if fault.statuscode != Fault_None then
            return (fault, TTWState UNKNOWN);

        walkstate.sdftype = AArch32.DecodeDescriptorTypeSD(descriptor, walkstate.level);
```

```
        case walkstate.sdftype of
            when SDFType_Invalid
                fault.domain     = domain;
                fault.statuscode = Fault_Translation;
                return (fault, TTWState UNKNOWN);

            when SDFType_Table
                domain = descriptor<8:5>;
                ns     = descriptor<3>;
                pxn    = descriptor<2>;

                walkstate.baseaddress.address = ZeroExtend(descriptor<31:10>:Zeros(10),
                                                           56);
                walkstate.level = 2;

            when SDFType_SmallPage
                nG  = descriptor<11>;
                s   = descriptor<10>;
                ap  = descriptor<9,5:4>;
                tex = descriptor<8:6>;
                c   = descriptor<3>;
                b   = descriptor<2>;
                xn  = descriptor<0>;

                walkstate.baseaddress.address = ZeroExtend(descriptor<31:12>:Zeros(12),
                                                           56);
                walkstate.istable = FALSE;

            when SDFType_LargePage
                xn  = descriptor<15>;
                tex = descriptor<14:12>;
                nG  = descriptor<11>;
                s   = descriptor<10>;
                ap  = descriptor<9,5:4>;
                c   = descriptor<3>;
                b   = descriptor<2>;

                walkstate.baseaddress.address = ZeroExtend(descriptor<31:16>:Zeros(16),
                                                           56);
                walkstate.istable = FALSE;

            when SDFType_Section
                ns     = descriptor<19>;
                nG     = descriptor<17>;
                s      = descriptor<16>;
                ap     = descriptor<15,11:10>;
                tex    = descriptor<14:12>;
                domain = descriptor<8:5>;
                xn     = descriptor<4>;
                c      = descriptor<3>;
                b      = descriptor<2>;
                pxn    = descriptor<0>;

                walkstate.baseaddress.address = ZeroExtend(descriptor<31:20>:Zeros(20),
                                                           56);
                walkstate.istable = FALSE;

            when SDFType_Supersection
                ns     = descriptor<19>;
                nG     = descriptor<17>;
                s      = descriptor<16>;
                ap     = descriptor<15,11:10>;
                tex    = descriptor<14:12>;
                xn     = descriptor<4>;
                c      = descriptor<3>;
                b      = descriptor<2>;
                pxn    = descriptor<0>;
                domain = '0000';
```

```
                walkstate.baseaddress.address = ZeroExtend(descriptor<8:5,23:20,31:24>:Zeros(24),
                                                           56);
                walkstate.istable = FALSE;

    until walkstate.sdftype != SDFType_Table;

    if afe == '1' && ap<0> == '0' then
        fault.domain     = domain;
        fault.statuscode = Fault_AccessFlag;
        return (fault, TTWState UNKNOWN);

    // Decode the TEX, C, B and S bits to produce target memory attributes
    if tre == '1' then
        walkstate.memattrs = AArch32.RemappedTEXDecode(regime, tex, c, b, s);
    elsif RemapRegsHaveResetValues() then
        walkstate.memattrs = AArch32.DefaultTEXDecode(tex, c, b, s);
    else
        walkstate.memattrs = MemoryAttributes IMPLEMENTATION_DEFINED;

    walkstate.permissions.ap  = ap;
    walkstate.permissions.xn  = xn;
    walkstate.permissions.pxn = pxn;
    walkstate.domain = domain;
    walkstate.nG     = nG;

    if accdesc.ss == SS_Secure && ns == '0' then
        walkstate.baseaddress.paspace = PAS_Secure;
    else
        walkstate.baseaddress.paspace = PAS_NonSecure;

    return (fault, walkstate);
```

Library pseudocode for aarch32/translation/walk/AArch32.S2IASize

```
// AArch32.S2IASize()
// ==================
// Retrieve the number of bits containing the input address for stage 2 translation

integer AArch32.S2IASize(bits(4) t0sz)
    return 32 - SInt(t0sz);
```

Library pseudocode for aarch32/translation/walk/AArch32.S2StartLevel

```
// AArch32.S2StartLevel()
// ======================
// Determine the initial lookup level when performing a stage 2 translation
// table walk

integer AArch32.S2StartLevel(bits(2) sl0)
    return 2 - UInt(sl0);
```

Library pseudocode for aarch32/translation/walk/AArch32.S2Walk

```
// AArch32.S2Walk()
// ================
// Traverse stage 2 translation tables in long format to obtain the final descriptor

(FaultRecord, TTWState) AArch32.S2Walk(FaultRecord fault_in, S2TTWParams walkparams,
                                       AccessDescriptor accdesc, AddressDescriptor ipa)
    FaultRecord fault = fault_in;

    if walkparams.sl0 IN {'1x'} || AArch32.S2InconsistentSL(walkparams) then
        fault.statuscode = Fault_Translation;
        fault.level      = 1;
        return (fault, TTWState UNKNOWN);

    // Input Address size
    iasize      = AArch32.S2IASize(walkparams.t0sz);
    startlevel  = AArch32.S2StartLevel(walkparams.sl0);
    levels      = FINAL_LEVEL - startlevel;
    granulebits = TGxGranuleBits(walkparams.tgx);
    stride      = granulebits - 3;

    if !IsZero(VTTBR<47:40>) then
        fault.statuscode = Fault_AddressSize;
        fault.level      = 0;
        return (fault, TTWState UNKNOWN);

    FullAddress baseaddress;
    baselsb = (iasize - (levels*stride + granulebits)) + 3;
    baseaddress.paspace = PAS_NonSecure;
    baseaddress.address = ZeroExtend(VTTBR<39:baselsb>:Zeros(baselsb), 56);

    TTWState walkstate;
    walkstate.baseaddress = baseaddress;
    walkstate.level       = startlevel;
    walkstate.istable     = TRUE;
    walkstate.memattrs    = WalkMemAttrs(walkparams.sh, walkparams.irgn,
                                         walkparams.orgn);

    indexmsb = iasize - 1;
    bits(64) descriptor;
    AccessDescriptor walkaccess = CreateAccDescS2TTW(accdesc);
    AddressDescriptor walkaddress;

    walkaddress.vaddress = ipa.vaddress;

    if HCR2.CD == '1' then
        walkaddress.memattrs = NormalNCMemAttr();
        walkaddress.memattrs.xs = walkstate.memattrs.xs;
    else
        walkaddress.memattrs = walkstate.memattrs;

    walkaddress.memattrs.shareability = EffectiveShareability(walkaddress.memattrs);

    integer indexlsb;
    DescriptorType desctype;
    repeat
        fault.level = walkstate.level;

        indexlsb = (FINAL_LEVEL - walkstate.level)*stride + granulebits;
        bits(40) index = ZeroExtend(ipa.paddress.address<indexmsb:indexlsb>:'000', 40);

        walkaddress.paddress.address = walkstate.baseaddress.address OR ZeroExtend(index, 56);
        walkaddress.paddress.paspace = walkstate.baseaddress.paspace;

        (fault, descriptor) = FetchDescriptor(walkparams.ee, walkaddress, walkaccess, fault, 64);

        if fault.statuscode != Fault_None then
            return (fault, TTWState UNKNOWN);

        desctype = AArch32.DecodeDescriptorTypeLD(descriptor, walkstate.level);
```

```
        case desctype of
            when DescriptorType_Table
                if !IsZero(descriptor<47:40>) then
                    fault.statuscode = Fault_AddressSize;
                    return (fault, TTWState UNKNOWN);

                walkstate.baseaddress.address = ZeroExtend(descriptor<39:12>:Zeros(12), 56);
                walkstate.level = walkstate.level + 1;
                indexmsb = indexlsb - 1;

            when DescriptorType_Invalid
                fault.statuscode = Fault_Translation;
                return (fault, TTWState UNKNOWN);

            when DescriptorType_Leaf
                walkstate.istable = FALSE;

    until desctype IN {DescriptorType_Leaf};

    // Check the output address is inside the supported range
    if !IsZero(descriptor<47:40>) then
        fault.statuscode = Fault_AddressSize;
        return (fault, TTWState UNKNOWN);

    // Check the access flag
    if descriptor<10> == '0' then
        fault.statuscode = Fault_AccessFlag;
        return (fault, TTWState UNKNOWN);

    // Unpack the descriptor into address and upper and lower block attributes
    walkstate.baseaddress.address = ZeroExtend(descriptor<39:indexlsb>:Zeros(indexlsb), 56);

    walkstate.permissions.s2ap = descriptor<7:6>;
    walkstate.permissions.s2xn = descriptor<54>;
    if HaveExtendedExecuteNeverExt() then
        walkstate.permissions.s2xnx = descriptor<53>;
    else
        walkstate.permissions.s2xnx = '0';

    memattr = descriptor<5:2>;
    sh      = descriptor<9:8>;
    s2aarch64 = FALSE;
    walkstate.memattrs   = S2DecodeMemAttrs(memattr, sh, s2aarch64);
    walkstate.contiguous = descriptor<52>;

    return (fault, walkstate);
```

Library pseudocode for aarch32/translation/walk/AArch32.TranslationSizeSD

```
// AArch32.TranslationSizeSD()
// ===========================
// Determine the size of the translation

integer AArch32.TranslationSizeSD(SDFType sdftype)
    integer tsize;
    case sdftype of
        when SDFType_SmallPage      tsize = 12;
        when SDFType_LargePage      tsize = 16;
        when SDFType_Section        tsize = 20;
        when SDFType_Supersection   tsize = 24;

    return tsize;
```

Library pseudocode for aarch32/translation/walk/RemapRegsHaveResetValues

```
// RemapRegsHaveResetValues()
// ==========================

boolean RemapRegsHaveResetValues();
```

Library pseudocode for aarch32/translation/walkparams/AArch32.GetS1TTWParams

```
// AArch32.GetS1TTWParams()
// ========================
// Returns stage 1 translation table walk parameters from respective controlling
// System registers.

S1TTWParams AArch32.GetS1TTWParams(Regime regime, bits(32) va)
    S1TTWParams walkparams;

    case regime of
        when Regime_EL2  walkparams = AArch32.S1TTWParamsEL2();
        when Regime_EL10 walkparams = AArch32.S1TTWParamsEL10(va);
        when Regime_EL30 walkparams = AArch32.S1TTWParamsEL30(va);

    return walkparams;
```

Library pseudocode for aarch32/translation/walkparams/AArch32.GetS2TTWParams

```
// AArch32.GetS2TTWParams()
// ========================
// Gather walk parameters for stage 2 translation

S2TTWParams AArch32.GetS2TTWParams()
    S2TTWParams walkparams;

    walkparams.tgx  = TGx_4KB;
    walkparams.s    = VTCR.S;
    walkparams.t0sz = VTCR.T0SZ;
    walkparams.sl0  = VTCR.SL0;
    walkparams.irgn = VTCR.IRGN0;
    walkparams.orgn = VTCR.ORGN0;
    walkparams.sh   = VTCR.SH0;
    walkparams.ee   = HSCTLR.EE;
    walkparams.ptw  = HCR.PTW;
    walkparams.vm   = HCR.VM OR HCR.DC;

    // VTCR.S must match VTCR.T0SZ[3]
    if walkparams.s != walkparams.t0sz<3> then
        (-, walkparams.t0sz) = ConstrainUnpredictableBits(Unpredictable_RESVTCRS, 4);

    return walkparams;
```

Library pseudocode for aarch32/translation/walkparams/AArch32.GetVARange

```
// AArch32.GetVARange()
// ====================
// Select the translation base address for stage 1 long-descriptor walks

VARange AArch32.GetVARange(bits(32) va, bits(3) t0sz, bits(3) t1sz)
    // Lower range Input Address size
    lo_iasize = AArch32.S1IASize(t0sz);
    // Upper range Input Address size
    up_iasize = AArch32.S1IASize(t1sz);

    if t1sz == '000' && t0sz == '000' then
        return VARange_LOWER;
    elsif t1sz == '000' then
        return if IsZero(va<31:lo_iasize>) then VARange_LOWER else VARange_UPPER;
    elsif t0sz == '000' then
        return if IsOnes(va<31:up_iasize>) then VARange_UPPER else VARange_LOWER;
    elsif IsZero(va<31:lo_iasize>) then
        return VARange_LOWER;
    elsif IsOnes(va<31:up_iasize>) then
        return VARange_UPPER;
    else
        // Will be reported as a Translation Fault
        return VARange UNKNOWN;
```

Library pseudocode for aarch32/translation/walkparams/AArch32.S1DCacheEnabled

```
// AArch32.S1DCacheEnabled()
// =========================
// Determine cacheability of stage 1 data accesses

boolean AArch32.S1DCacheEnabled(Regime regime)
    case regime of
        when Regime_EL30 return SCTLR_S.C == '1';
        when Regime_EL2  return HSCTLR.C == '1';
        when Regime_EL10 return (if HaveAArch32EL(EL3) then SCTLR_NS.C else SCTLR.C) == '1';
```

Library pseudocode for aarch32/translation/walkparams/AArch32.S1ICacheEnabled

```
// AArch32.S1ICacheEnabled()
// =========================
// Determine cacheability of stage 1 instruction fetches

boolean AArch32.S1ICacheEnabled(Regime regime)
    case regime of
        when Regime_EL30 return SCTLR_S.I == '1';
        when Regime_EL2  return HSCTLR.I == '1';
        when Regime_EL10 return (if HaveAArch32EL(EL3) then SCTLR_NS.I else SCTLR.I) == '1';
```

Library pseudocode for aarch32/translation/walkparams/AArch32.S1TTWParamsEL10

```
// AArch32.S1TTWParamsEL10()
// ==========================
// Gather stage 1 translation table walk parameters for EL1&0 regime
// (with EL2 enabled or disabled).

S1TTWParams AArch32.S1TTWParamsEL10(bits(32) va)
    bits(64) mair;
    bit sif;
    TTBCR_Type ttbcr;
    TTBCR2_Type ttbcr2;
    SCTLR_Type sctlr;

    if ELUsingAArch32(EL3) then
        ttbcr  = TTBCR_NS;
        ttbcr2 = TTBCR2_NS;
        sctlr  = SCTLR_NS;
        mair   = MAIR1_NS:MAIR0_NS;
        sif    = SCR.SIF;
    else
        ttbcr  = TTBCR;
        ttbcr2 = TTBCR2;
        sctlr  = SCTLR;
        mair   = MAIR1:MAIR0;
        sif    = if HaveEL(EL3) then SCR_EL3.SIF else '0';

    assert ttbcr.EAE == '1';
    S1TTWParams walkparams;

    walkparams.t0sz   = ttbcr.T0SZ;
    walkparams.t1sz   = ttbcr.T1SZ;
    walkparams.ee     = sctlr.EE;
    walkparams.wxn    = sctlr.WXN;
    walkparams.uwxn   = sctlr.UWXN;
    walkparams.ntlsmd = if HaveTrapLoadStoreMultipleDeviceExt() then sctlr.nTLSMD else '1';
    walkparams.mair   = mair;
    walkparams.sif    = sif;

    varange = AArch32.GetVARange(va, walkparams.t0sz, walkparams.t1sz);
    if varange == VARange_LOWER then
        walkparams.sh   = ttbcr.SH0;
        walkparams.irgn = ttbcr.IRGN0;
        walkparams.orgn = ttbcr.ORGN0;
        walkparams.hpd  = if AArch32.HaveHPDExt() then ttbcr.T2E AND ttbcr2.HPD0 else '0';
    else
        walkparams.sh   = ttbcr.SH1;
        walkparams.irgn = ttbcr.IRGN1;
        walkparams.orgn = ttbcr.ORGN1;
        walkparams.hpd  = if AArch32.HaveHPDExt() then ttbcr.T2E AND ttbcr2.HPD1 else '0';

    return walkparams;
```

Library pseudocode for aarch32/translation/walkparams/AArch32.S1TTWParamsEL2

```
// AArch32.S1TTWParamsEL2()
// ========================
// Gather stage 1 translation table walk parameters for EL2 regime

S1TTWParams AArch32.S1TTWParamsEL2()
    S1TTWParams walkparams;

    walkparams.tgx  = TGx_4KB;
    walkparams.t0sz = HTCR.T0SZ;
    walkparams.irgn = HTCR.SH0;
    walkparams.orgn = HTCR.IRGN0;
    walkparams.sh   = HTCR.ORGN0;
    walkparams.hpd  = if AArch32.HaveHPDExt() then HTCR.HPD else '0';
    walkparams.ee   = HSCTLR.EE;
    walkparams.wxn  = HSCTLR.WXN;
    if HaveTrapLoadStoreMultipleDeviceExt() then
        walkparams.ntlsmd = HSCTLR.nTLSMD;
    else
        walkparams.ntlsmd = '1';

    walkparams.mair = HMAIR1:HMAIR0;

    return walkparams;
```

Library pseudocode for aarch32/translation/walkparams/AArch32.S1TTWParamsEL30

```
// AArch32.S1TTWParamsEL30()
// =========================
// Gather stage 1 translation table walk parameters for EL3&0 regime

S1TTWParams AArch32.S1TTWParamsEL30(bits(32) va)
    assert TTBCR_S.EAE == '1';
    S1TTWParams walkparams;

    walkparams.t0sz   = TTBCR_S.T0SZ;
    walkparams.t1sz   = TTBCR_S.T1SZ;
    walkparams.ee     = SCTLR_S.EE;
    walkparams.wxn    = SCTLR_S.WXN;
    walkparams.uwxn   = SCTLR_S.UWXN;
    walkparams.ntlsmd = if HaveTrapLoadStoreMultipleDeviceExt() then SCTLR_S.nTLSMD else '1';
    walkparams.mair   = MAIR1_S:MAIR0_S;
    walkparams.sif    = SCR.SIF;

    varange = AArch32.GetVARange(va, walkparams.t0sz, walkparams.t1sz);
    if varange == VARange_LOWER then
        walkparams.sh   = TTBCR_S.SH0;
        walkparams.irgn = TTBCR_S.IRGN0;
        walkparams.orgn = TTBCR_S.ORGN0;
        walkparams.hpd  = if AArch32.HaveHPDExt() then TTBCR_S.T2E AND TTBCR2_S.HPD0 else '0';
    else
        walkparams.sh   = TTBCR_S.SH1;
        walkparams.irgn = TTBCR_S.IRGN1;
        walkparams.orgn = TTBCR_S.ORGN1;
        walkparams.hpd  = if AArch32.HaveHPDExt() then TTBCR_S.T2E AND TTBCR2_S.HPD1 else '0';

    return walkparams;
```

Library pseudocode for aarch64/debug/brbe/BRBCycleCountingEnabled

```
// BRBCycleCountingEnabled()
// =========================
// Returns TRUE if the recording of cycle counts is allowed,
// FALSE otherwise.

boolean BRBCycleCountingEnabled()
    if HaveEL(EL2) && BRBCR_EL2.CC == '0' then return FALSE;
    if BRBCR_EL1.CC == '0' then return FALSE;
    return TRUE;
```

Library pseudocode for aarch64/debug/brbe/BRBEBranch

```
// BRBEBranch()
// ============
// Called to write branch record for the following branches when BRB is active:
// direct branches,
// indirect branches,
// direct branches with link,
// indirect branches with link,
// returns from subroutines.

BRBEBranch(BranchType br_type, boolean cond, bits(64) target_address)
    if BranchRecordAllowed(PSTATE.EL) && FilterBranchRecord(br_type, cond) then
        bits(6) branch_type;
        case br_type of
            when BranchType_DIR
                branch_type = if cond then '001000' else '000000';
            when BranchType_INDIR      branch_type = '000001';
            when BranchType_DIRCALL    branch_type = '000010';
            when BranchType_INDCALL    branch_type = '000011';
            when BranchType_RET        branch_type = '000101';
            otherwise                  Unreachable();

        bit ccu;
        bits(14) cc;
        (ccu, cc) = BranchEncCycleCount();
        bit lastfailed = if HaveTME() then BRBFCR_EL1.LASTFAILED else '0';
        bit transactional = if HaveTME() && TSTATE.depth > 0 then '1' else '0';
        bits(2) el = PSTATE.EL;
        bit mispredict = if BRBEMispredictAllowed() && BranchMispredict() then '1' else '0';

        UpdateBranchRecordBuffer(ccu, cc, lastfailed, transactional, branch_type, el, mispredict,
                                 '11', PC[], target_address);

        BRBFCR_EL1.LASTFAILED = '0';

        PMUEvent(PMU_EVENT_BRB_FILTRATE);

    return;
```

Library pseudocode for aarch64/debug/brbe/BRBEBranchOnISB

```
// BRBEBranchOnISB()
// =================
// Returns TRUE if ISBs generate Branch records, and FALSE otherwise.

boolean BRBEBranchOnISB()
    return boolean IMPLEMENTATION_DEFINED "ISB generates Branch records";
```

Library pseudocode for aarch64/debug/brbe/BRBEDebugStateExit

```
// BRBEDebugStateExit()
// ====================
// Called to write Debug state exit branch record when BRB is active.

BRBEDebugStateExit(bits(64) target_address)
    if BranchRecordAllowed(PSTATE.EL) then
        // Debug state is a prohibited region, therefore ccu=1, cc=0, source_address=0
        bits(6) branch_type = '111001';
        bit ccu = '1';
        bits(14) cc = Zeros(14);
        bit lastfailed = if HaveTME() then BRBFCR_EL1.LASTFAILED else '0';
        bit transactional = '0';
        bits(2) el = PSTATE.EL;
        bit mispredict = '0';

        UpdateBranchRecordBuffer(ccu, cc, lastfailed, transactional, branch_type, el, mispredict,
                                 '01', Zeros(64), target_address);

        BRBFCR_EL1.LASTFAILED = '0';

        PMUEvent(PMU_EVENT_BRB_FILTRATE);

    return;
```

Library pseudocode for aarch64/debug/brbe/BRBEException

```
// BRBEException()
// ===============
// Called to write exception branch record when BRB is active.

BRBEException(ExceptionRecord erec, bits(64) preferred_exception_return,
              bits(64) target_address_in, bits(2) target_el, boolean trappedsyscallinst)
    bits(64) target_address = target_address_in;
    Exception except = erec.exceptype;
    bits(25) iss = erec.syndrome;
    case target_el of
        when EL3  if !HaveBRBEv1p1() || (MDCR_EL3.E3BREC == MDCR_EL3.E3BREW) then return;
        when EL2  if BRBCR_EL2.EXCEPTION == '0' then return;
        when EL1  if BRBCR_EL1.EXCEPTION == '0' then return;

    boolean source_valid = BranchRecordAllowed(PSTATE.EL);
    boolean target_valid = BranchRecordAllowed(target_el);

    if source_valid || target_valid then
        bits(6) branch_type;
        case except of
            when Exception_Uncategorized         branch_type = '100011'; // Trap
            when Exception_WFxTrap               branch_type = '100011'; // Trap
            when Exception_CP15RTTrap            branch_type = '100011'; // Trap
            when Exception_CP15RRTTrap           branch_type = '100011'; // Trap
            when Exception_CP14RTTrap            branch_type = '100011'; // Trap
            when Exception_CP14DTTrap            branch_type = '100011'; // Trap
            when Exception_AdvSIMDFPAccessTrap   branch_type = '100011'; // Trap
            when Exception_FPIDTrap              branch_type = '100011'; // Trap
            when Exception_PACTrap               branch_type = '100011'; // Trap
            when Exception_TSTARTAccessTrap      branch_type = '100011'; // Trap
            when Exception_CP14RRTTrap           branch_type = '100011'; // Trap
            when Exception_BranchTarget          branch_type = '101011'; // Inst Fault
            when Exception_IllegalState          branch_type = '100011'; // Trap
            when Exception_SupervisorCall
                if !trappedsyscallinst then      branch_type = '100010'; // Call
                else                             branch_type = '100011'; // Trap
            when Exception_HypervisorCall        branch_type = '100010'; // Call
            when Exception_MonitorCall
                if !trappedsyscallinst then      branch_type = '100010'; // Call
                else                             branch_type = '100011'; // Trap
            when Exception_SystemRegisterTrap    branch_type = '100011'; // Trap
            when Exception_SystemRegister128Trap branch_type = '100011'; // Trap
            when Exception_SVEAccessTrap         branch_type = '100011'; // Trap
            when Exception_SMEAccessTrap         branch_type = '100011'; // Trap
            when Exception_ERetTrap              branch_type = '100011'; // Trap
            when Exception_PACFail               branch_type = '101100'; // Data Fault
            when Exception_InstructionAbort      branch_type = '101011'; // Inst Fault
            when Exception_PCAlignment           branch_type = '101010'; // Alignment
            when Exception_DataAbort             branch_type = '101100'; // Data Fault
            when Exception_NV2DataAbort          branch_type = '101100'; // Data Fault
            when Exception_SPAlignment           branch_type = '101010'; // Alignment
            when Exception_FPTrappedException    branch_type = '100011'; // Trap
            when Exception_SError                branch_type = '100100'; // System Error
            when Exception_Breakpoint            branch_type = '100110'; // Inst debug
            when Exception_SoftwareStep          branch_type = '100110'; // Inst debug
            when Exception_Watchpoint            branch_type = '100111'; // Data debug
            when Exception_NV2Watchpoint         branch_type = '100111'; // Data debug
            when Exception_SoftwareBreakpoint    branch_type = '100110'; // Inst debug
            when Exception_IRQ                   branch_type = '101110'; // IRQ
            when Exception_FIQ                   branch_type = '101111'; // FIQ
            when Exception_MemCpyMemSet          branch_type = '100011'; // Trap
            when Exception_GCSFail
                if iss<23:20> == '0000' then     branch_type = '101100'; // Data Fault
                elsif iss<23:20> == '0001' then  branch_type = '101011'; // Inst Fault
                elsif iss<23:20> == '0010' then  branch_type = '100011'; // Trap
                else                             Unreachable();
            otherwise                            Unreachable();

        bit ccu;
        bits(14) cc;
```

```
        (ccu, cc) = BranchEncCycleCount();
        bit lastfailed = if HaveTME() then BRBFCR_EL1.LASTFAILED else '0';
        bit transactional = if source_valid && HaveTME() && TSTATE.depth > 0 then '1' else '0';
        bits(2) el = if target_valid then target_el else '00';
        bit mispredict = '0';
        bit sv = if source_valid then '1' else '0';
        bit tv = if target_valid then '1' else '0';
        bits(64) source_address = if source_valid then preferred_exception_return else Zeros(64);

        if !target_valid then
            target_address = Zeros(64);
        else
            target_address = AArch64.BranchAddr(target_address, target_el);

        UpdateBranchRecordBuffer(ccu, cc, lastfailed, transactional, branch_type, el, mispredict,
                                 sv:tv, source_address, target_address);

        BRBFCR_EL1.LASTFAILED = '0';

        PMUEvent(PMU_EVENT_BRB_FILTRATE);

    return;
```

Library pseudocode for aarch64/debug/brbe/BRBEExceptionReturn

```
// BRBEExceptionReturn()
// =====================
// Called to write exception return branch record when BRB is active.

BRBEExceptionReturn(bits(64) target_address_in, bits(2) source_el)
    bits(64) target_address = target_address_in;
    case source_el of
        when EL3  if !HaveBRBEv1p1() || (MDCR_EL3.E3BREC == MDCR_EL3.E3BREW) then return;
        when EL2  if BRBCR_EL2.ERTN == '0' then return;
        when EL1  if BRBCR_EL1.ERTN == '0' then return;

    boolean source_valid = BranchRecordAllowed(source_el);
    boolean target_valid = BranchRecordAllowed(PSTATE.EL);

    if source_valid || target_valid then
        bits(6) branch_type = '000111';
        bit ccu;
        bits(14) cc;
        (ccu, cc) = BranchEncCycleCount();
        bit lastfailed = if HaveTME() then BRBFCR_EL1.LASTFAILED else '0';
        bit transactional = if source_valid && HaveTME() && TSTATE.depth > 0 then '1' else '0';
        bits(2) el = if target_valid then PSTATE.EL else '00';
        bit mispredict = if (source_valid && BRBEMispredictAllowed() &&
                             BranchMispredict()) then '1' else '0';
        bit sv = if source_valid then '1' else '0';
        bit tv = if target_valid then '1' else '0';
        bits(64) source_address = if source_valid then PC[] else Zeros(64);
        if !target_valid then
            target_address = Zeros(64);

        UpdateBranchRecordBuffer(ccu, cc, lastfailed, transactional, branch_type, el, mispredict,
                                 sv:tv, source_address, target_address);

        BRBFCR_EL1.LASTFAILED = '0';

        PMUEvent(PMU_EVENT_BRB_FILTRATE);

    return;
```

Library pseudocode for aarch64/debug/brbe/BRBEFreeze

```
// BRBEFreeze()
// ============
// Generates BRBE freeze event.

BRBEFreeze()
    BRBFCR_EL1.PAUSED = '1';
    BRBTS_EL1 = GetTimestamp(BRBETimeStamp());
```

Library pseudocode for aarch64/debug/brbe/BRBEISB

```
// BRBEISB()
// =========
// Handles ISB instruction for BRBE.

BRBEISB()
    boolean branch_conditional = FALSE;
    BRBEBranch(BranchType_DIR, branch_conditional, PC[] + 4);
```

Library pseudocode for aarch64/debug/brbe/BRBEMispredictAllowed

```
// BRBEMispredictAllowed()
// =======================
// Returns TRUE if the recording of branch misprediction is allowed,
// FALSE otherwise.

boolean BRBEMispredictAllowed()
    if HaveEL(EL2) && BRBCR_EL2.MPRED == '0' then return FALSE;
    if BRBCR_EL1.MPRED == '0' then return FALSE;
    return TRUE;
```

Library pseudocode for aarch64/debug/brbe/BRBETimeStamp

```
// BRBETimeStamp()
// ===============
// Returns captured timestamp.

TimeStamp BRBETimeStamp()
    if HaveEL(EL2) then
        TS_el2 = BRBCR_EL2.TS;
        if !HaveECVExt() && TS_el2 == '10' then
            // Reserved value
            (-, TS_el2) = ConstrainUnpredictableBits(Unpredictable_EL2TIMESTAMP, 2);
        case TS_el2 of
            when '00'
                // Falls out to check BRBCR_EL1.TS
            when '01'
                return TimeStamp_Virtual;
            when '10'
                assert HaveECVExt(); // Otherwise ConstrainUnpredictableBits removes this case
                return TimeStamp_OffsetPhysical;
            when '11'
                return TimeStamp_Physical;

    TS_el1 = BRBCR_EL1.TS;
    if TS_el1 == '00' || (!HaveECVExt() && TS_el1 == '10') then
        // Reserved value
        (-, TS_el1) = ConstrainUnpredictableBits(Unpredictable_EL1TIMESTAMP, 2);
    case TS_el1 of
        when '01'
            return TimeStamp_Virtual;
        when '10'
            return TimeStamp_OffsetPhysical;
        when '11'
            return TimeStamp_Physical;
        otherwise
            Unreachable();           // ConstrainUnpredictableBits removes this case
```

Library pseudocode for aarch64/debug/brbe/BRB_IALL

```
// BRB_IALL()
// ==========
// Called to perform invalidation of branch records

BRB_IALL()
    for i = 0 to GetBRBENumRecords() - 1
        Records_SRC[i] = Zeros(64);
        Records_TGT[i] = Zeros(64);
        Records_INF[i] = Zeros(64);
```

Library pseudocode for aarch64/debug/brbe/BRB_INJ

```
// BRB_INJ()
// =========
// Called to perform manual injection of branch records.

BRB_INJ()
    UpdateBranchRecordBuffer(BRBINFINJ_EL1.CCU, BRBINFINJ_EL1.CC, BRBINFINJ_EL1.LASTFAILED,
                             BRBINFINJ_EL1.T, BRBINFINJ_EL1.TYPE, BRBINFINJ_EL1.EL,
                             BRBINFINJ_EL1.MPRED, BRBINFINJ_EL1.VALID, BRBSRCINJ_EL1.ADDRESS,
                             BRBTGTINJ_EL1.ADDRESS);
    BRBINFINJ_EL1 = bits(64) UNKNOWN;
    BRBSRCINJ_EL1 = bits(64) UNKNOWN;
    BRBTGTINJ_EL1 = bits(64) UNKNOWN;

    if ConstrainUnpredictableBool(Unpredictable_BRBFILTRATE) then PMUEvent(PMU_EVENT_BRB_FILTRATE);
```

Library pseudocode for aarch64/debug/brbe/Branch

```
type BRBSRCType;
type BRBTGTType;
type BRBINFType;
```

Library pseudocode for aarch64/debug/brbe/BranchEncCycleCount

```
// BranchEncCycleCount()
// =====================
// The first return result is '1' if either of the following is true, and '0' otherwise:
// - This is the first Branch record after the PE exited a Prohibited Region.
// - This is the first Branch record after cycle counting has been enabled.
// If the first return return is '0', the second return result is the encoded cycle count
// since the last branch.
// The format of this field uses a mantissa and exponent to express the cycle count value.
//  - bits[7:0] indicate the mantissa M.
//  - bits[13:8] indicate the exponent E.
// The cycle count is expressed using the following function:
//   cycle_count = (if IsZero(E) then UInt(M) else UInt('1':M:Zeros(UInt(E)-1)))
// A value of all ones in both the mantissa and exponent indicates the cycle count value
// exceeded the size of the cycle counter.
// If the cycle count is not known, the second return result is zero.

(bit, bits(14)) BranchEncCycleCount();
```

Library pseudocode for aarch64/debug/brbe/BranchMispredict

```
// BranchMispredict()
// ==================
// Returns TRUE if the branch being executed was mispredicted, FALSE otherwise.

boolean BranchMispredict();
```

Library pseudocode for aarch64/debug/brbe/BranchRawCycleCount

```
// BranchRawCycleCount()
// =====================
// If the cycle count is known, the return result is the cycle count since the last branch.

integer BranchRawCycleCount();
```

Library pseudocode for aarch64/debug/brbe/BranchRecordAllowed

```
// BranchRecordAllowed()
// =====================
// Returns TRUE if branch recording is allowed, FALSE otherwise.

boolean BranchRecordAllowed(bits(2) el)
    if ELUsingAArch32(el) then
        return FALSE;

    if BRBFCR_EL1.PAUSED == '1' then
        return FALSE;

    if el == EL3 && HaveBRBEv1p1() then
        return (MDCR_EL3.E3BREC != MDCR_EL3.E3BREW);

    if HaveEL(EL3) && (MDCR_EL3.SBRBE == '00' ||
        (CurrentSecurityState() == SS_Secure && MDCR_EL3.SBRBE == '01')) then
        return FALSE;

    case el of
        when EL3  return FALSE;                // FEAT_BRBEv1p1 not implemented
        when EL2  return BRBCR_EL2.E2BRE == '1';
        when EL1  return BRBCR_EL1.E1BRE == '1';
        when EL0
            if EL2Enabled() && HCR_EL2.TGE == '1' then
                return BRBCR_EL2.E0HBRE == '1';
            else
                return BRBCR_EL1.E0BRE == '1';
```

Library pseudocode for aarch64/debug/brbe/Contents

```
// Contents of the Branch Record Buffer
//=====================================

array [0..63] of BRBSRCType Records_SRC;

array [0..63] of BRBTGTType Records_TGT;

array [0..63] of BRBINFType Records_INF;
```

Library pseudocode for aarch64/debug/brbe/FilterBranchRecord

```
// FilterBranchRecord()
// ====================
// Returns TRUE if the branch record is not filtered out, FALSE otherwise.

boolean FilterBranchRecord(BranchType br, boolean cond)
    case br of
        when BranchType_DIRCALL
            return BRBFCR_EL1.DIRCALL != BRBFCR_EL1.EnI;
        when BranchType_INDCALL
            return BRBFCR_EL1.INDCALL != BRBFCR_EL1.EnI;
        when BranchType_RET
            return BRBFCR_EL1.RTN != BRBFCR_EL1.EnI;
        when BranchType_DIR
            if cond then
                return BRBFCR_EL1.CONDDIR != BRBFCR_EL1.EnI;
            else
                return BRBFCR_EL1.DIRECT != BRBFCR_EL1.EnI;
        when BranchType_INDIR
            return BRBFCR_EL1.INDIRECT != BRBFCR_EL1.EnI;
        otherwise  Unreachable();
    return FALSE;
```

Library pseudocode for aarch64/debug/brbe/FirstBranchAfterProhibited

```
// FirstBranchAfterProhibited()
// ============================
// Returns TRUE if branch recorded is the first branch after a prohibited region,
// FALSE otherwise.

FirstBranchAfterProhibited();
```

Library pseudocode for aarch64/debug/brbe/GetBRBENumRecords

```
// GetBRBENumRecords()
// ===================
// Returns the number of branch records implemented.

integer GetBRBENumRecords()
    assert UInt(BRBIDR0_EL1.NUMREC) IN {0x08, 0x10, 0x20, 0x40};
    return integer IMPLEMENTATION_DEFINED "Number of BRB records";
```

Library pseudocode for aarch64/debug/brbe/Getter

```
// Getter functions for branch records
// ===================================
// Functions used by MRS instructions that access branch records

BRBSRCType BRBSRC_EL1[integer n]
    assert n IN {0..31};
    integer record = UInt(BRBFCR_EL1.BANK:n<4:0>);
    if record < GetBRBENumRecords() then
        return Records_SRC[record];
    else
        return Zeros(64);

BRBTGTType BRBTGT_EL1[integer n]
    assert n IN {0..31};
    integer record = UInt(BRBFCR_EL1.BANK:n<4:0>);
    if record < GetBRBENumRecords() then
        return Records_TGT[record];
    else
        return Zeros(64);

BRBINFType BRBINF_EL1[integer n]
    assert n IN {0..31};
    integer record = UInt(BRBFCR_EL1.BANK:n<4:0>);
    if record < GetBRBENumRecords() then
        return Records_INF[record];
    else
        return Zeros(64);
```

Library pseudocode for aarch64/debug/brbe/ShouldBRBEFreeze

```
// ShouldBRBEFreeze()
// ==================
// Returns TRUE if the BRBE freeze event conditions have been met, and FALSE otherwise.

boolean ShouldBRBEFreeze()
    if !BranchRecordAllowed(PSTATE.EL) then return FALSE;
    boolean check_e      = FALSE;
    boolean check_cnten  = FALSE;
    boolean check_inten  = FALSE;
    boolean exclude_sync = FALSE;
    boolean exclude_cyc  = TRUE;
    boolean include_lo;
    boolean include_hi;

    if HaveEL(EL2) then
        include_lo = (BRBCR_EL1.FZP == '1');
        include_hi = (BRBCR_EL2.FZP == '1');
    else
        include_lo = TRUE;
        include_hi = TRUE;

    return PMUOverflowCondition(check_e, check_cnten, check_inten,
                                include_hi, include_lo, exclude_cyc,
                                exclude_sync);
```

Library pseudocode for aarch64/debug/brbe/UpdateBranchRecordBuffer

```
// UpdateBranchRecordBuffer()
// ==========================
// Add a new Branch record to the buffer.

UpdateBranchRecordBuffer(bit ccu, bits(14) cc, bit lastfailed, bit transactional,
                         bits(6) branch_type, bits(2) el, bit mispredict, bits(2) valid,
                         bits(64) source_address, bits(64) target_address)
    // Shift the Branch Records in the buffer
    for i = GetBRBENumRecords() - 1 downto 1
        Records_SRC[i] = Records_SRC[i - 1];
        Records_TGT[i] = Records_TGT[i - 1];
        Records_INF[i] = Records_INF[i - 1];

    Records_INF[0].CCU        = ccu;
    Records_INF[0].CC         = cc;

    Records_INF[0].EL         = el;
    Records_INF[0].VALID      = valid;
    Records_INF[0].T          = transactional;
    Records_INF[0].LASTFAILED = lastfailed;
    Records_INF[0].MPRED      = mispredict;
    Records_INF[0].TYPE       = branch_type;

    Records_SRC[0] = source_address;
    Records_TGT[0] = target_address;

    return;
```

Library pseudocode for aarch64/debug/breakpoint/AArch64.BreakpointMatch

```
// AArch64.BreakpointMatch()
// =========================
// Breakpoint matching in an AArch64 translation regime.
// Returns a pair of booleans, the first indicates if the match was successful and the second if
// the first value should be inverted because the breakpoint is configured for a mismatch.

(boolean, boolean) AArch64.BreakpointMatch(integer n, bits(64) vaddress, AccessDescriptor accdesc,
                                           integer size)
    assert !ELUsingAArch32(S1TranslationRegime());
    assert n < NumBreakpointsImplemented();

    linking_enabled = (DBGBCR_EL1[n].BT IN {'0x11', '1xx1'} ||
                       (HaveFeatABLE() && DBGBCR_EL1[n].BT2 == '1'));

    // A breakpoint that has linking enabled does not generate debug events in isolation
    if linking_enabled then
        return (FALSE, FALSE);

    enabled    = IsBreakpointEnabled(n);
    linked     = DBGBCR_EL1[n].BT IN {'0x01'};
    isbreakpnt = TRUE;
    linked_to  = FALSE;
    lbnx      = if Havev8p9Debug() then DBGBCR_EL1[n].LBNX else '00';
    linked_n = UInt(lbnx : DBGBCR_EL1[n].LBN);
    ssce = if HaveRME() then DBGBCR_EL1[n].SSCE else '0';
    state_match = AArch64.StateMatch(DBGBCR_EL1[n].SSC, ssce, DBGBCR_EL1[n].HMC,
                                     DBGBCR_EL1[n].PMC, linked, linked_n, isbreakpnt,
                                     vaddress, accdesc);

    (value_match, valid_mismatch) = AArch64.BreakpointValueMatch(n, vaddress, linked_to,
                                                                 isbreakpnt);

    if HaveAArch32() && size == 4 then                    // Check second halfword
        // If the breakpoint address and BAS of an Address breakpoint match the address of the
        // second halfword of an instruction, but not the address of the first halfword, it is
        // CONSTRAINED UNPREDICTABLE whether or not this breakpoint generates a Breakpoint debug
        // event.
        (match_i, -) = AArch64.BreakpointValueMatch(n, vaddress + 2, linked_to,
                                                    isbreakpnt);
        if !value_match && match_i then
            value_match = ConstrainUnpredictableBool(Unpredictable_BPMATCHHALF);

    if vaddress<1> == '1' && DBGBCR_EL1[n].BAS == '1111' then
        // The above notwithstanding, if DBGBCR_EL1[n].BAS == '1111', then it is CONSTRAINED
        // UNPREDICTABLE whether or not a Breakpoint debug event is generated for an instruction
        // at the address DBGBVR_EL1[n]+2.
        if value_match then value_match = ConstrainUnpredictableBool(Unpredictable_BPMATCHHALF);

    match = value_match && state_match && enabled;
    is_mismatch = valid_mismatch && state_match && enabled;

    return (match, is_mismatch);
```

Library pseudocode for aarch64/debug/breakpoint/AArch64.BreakpointValueMatch

```
// AArch64.BreakpointValueMatch()
// ==============================
// Returns a pair of booleans, the first indicates if the value match was successful and the
// second if the first value should be inverted for a mismatch.

(boolean, boolean) AArch64.BreakpointValueMatch(integer n_in, bits(64) vaddress,
                                                boolean linked_to, boolean isbreakpnt)

    // "n" is the identity of the breakpoint unit to match against.
    // "vaddress" is the current instruction address, ignored if linked_to is TRUE and for Context
    //   matching breakpoints.
    // "linked_to" is TRUE if this is a call from StateMatch for linking.
    // "isbreakpnt"  TRUE is this is a call from BreakpointMatch or from StateMatch for a
    // linked breakpoint.
    integer n = n_in;
    Constraint c;

    // If a non-existent breakpoint then it is CONSTRAINED UNPREDICTABLE whether this gives
    // no match or the breakpoint is mapped to another UNKNOWN implemented breakpoint.
    if n >= NumBreakpointsImplemented() then
        (c, n) = ConstrainUnpredictableInteger(0, NumBreakpointsImplemented() - 1,
                                               Unpredictable_BPNOTIMPL);
        assert c IN {Constraint_DISABLED, Constraint_UNKNOWN};
        if c == Constraint_DISABLED then return (FALSE, FALSE);

    // If this breakpoint is not enabled, it cannot generate a match.
    // (This could also happen on a call from StateMatch for linking).
    if !IsBreakpointEnabled(n) then return (FALSE, FALSE);

    // If BT is set to a reserved type, behaves either as disabled or as a not-reserved type.
    dbgtype = DBGBCR_EL1[n].BT;
    bt2 = if HaveFeatABLE() then DBGBCR_EL1[n].BT2 else '0';

    (c, bt2, dbgtype) = AArch64.ReservedBreakpointType(n, bt2, dbgtype);
    if c == Constraint_DISABLED then return (FALSE, FALSE);
    // Otherwise the value returned by ConstrainUnpredictableBits must be a not-reserved value

    // Determine what to compare against.
    match_addr      = (dbgtype IN {'0x0x'});
    mismatch        = (dbgtype IN {'010x'});
    match_vmid      = (dbgtype IN {'10xx'});
    match_cid       = (dbgtype IN {'001x'});
    match_cid1      = (dbgtype IN {'101x', 'x11x'});
    match_cid2      = (dbgtype IN {'11xx'});
    linking_enabled = (dbgtype IN {'xx11', '1xx1'} || bt2 == '1');

    // If this is a call from StateMatch, return FALSE if the breakpoint is not
    // programmed with linking enabled.
    if linked_to && !linking_enabled then
        return (FALSE, FALSE);

    // If called from BreakpointMatch return FALSE for Linked context ID and/or VMID matches.
    if !linked_to && linking_enabled && !match_addr then
        return (FALSE, FALSE);

    // If a linked breakpoint is linked to an address matching breakpoint,
    // the behavior is CONSTRAINED UNPREDICTABLE.
    if linked_to && match_addr && isbreakpnt then
        if !ConstrainUnpredictableBool(Unpredictable_BPLINKEDADDRMATCH) then
            return (FALSE, FALSE);

    // A breakpoint programmed for address mismatch does not match in AArch32 state.
    if mismatch && UsingAArch32() then
        return (FALSE, FALSE);

    boolean bvr_match  = FALSE;
    boolean bxvr_match = FALSE;
    integer mask;

    if HaveFeatABLE() then
```

```
        mask = UInt(DBGBCR_EL1[n].MASK);

        // If the mask is set to a reserved value, the behavior is CONSTRAINED UNPREDICTABLE.
        if mask IN {1, 2} then
            (c, mask) = ConstrainUnpredictableInteger(3, 31, Unpredictable_RESBPMASK);
            assert c IN {Constraint_DISABLED, Constraint_NONE, Constraint_UNKNOWN};
            case c of
                when Constraint_DISABLED return (FALSE, FALSE);  // Disabled
                when Constraint_NONE     mask = 0;               // No masking
                // Otherwise the value returned by ConstrainUnpredictableBits must
                // be a not-reserved value.

        if mask != 0 then
            // When DBGBCR_EL1[n].MASK is a valid nonzero value, the behavior is
            // CONSTRAINED UNPREDICTABLE if any of the following are true:
            //     - DBGBCR_EL1[n].<BT2,BT> is programmed for a Context matching breakpoint.
            //     - DBGBCR_EL1[n].BAS is not '1111' and AArch32 is supported at EL0.
            if ((match_cid || match_cid1 || match_cid2) ||
                (DBGBCR_EL1[n].BAS != '1111' && HaveAArch32())) then
                if !ConstrainUnpredictableBool(Unpredictable_BPMASK) then return (FALSE, FALSE);
        else
            // A stand-alone mismatch of a single address is not supported.
            if mismatch then
                return (FALSE, FALSE);

    else
        mask = 0;

    // Do the comparison.
    if match_addr then
        boolean byte_select_match;
        integer byte = UInt(vaddress<1:0>);

        if HaveAArch32() then
            // T32 instructions can be executed at EL0 in an AArch64 translation regime.
            assert byte IN {0,2};                        // "vaddress" is halfword aligned
            byte_select_match = (DBGBCR_EL1[n].BAS<byte> == '1');
        else
            assert byte == 0;                            // "vaddress" is word aligned
            byte_select_match = TRUE;                    // DBGBCR_EL1[n].BAS<byte> is RES1

        // When FEAT_LVA3 is not implemented, if the DBGBVR_EL1[n].RESS field bits are not a
        // sign extension of the MSB of DBGBVR_EL1[n].VA, it is UNPREDICTABLE whether they
        // appear to be included in the match.
        // If 'vaddress' is outside of the current virtual address space, then the access
        // generates a Translation fault.
        integer top    = DebugAddrTop();
        integer bottom = 2;
        if (top < 55 && !IsOnes(DBGBVR_EL1[n]<63:top>) && !IsZero(DBGBVR_EL1[n]<63:top>) &&
              ConstrainUnpredictableBool(Unpredictable_DBGxVR_RESS)) then
            top = 63;

        if mask > bottom then
            bvr_match = (vaddress<top:mask> == DBGBVR_EL1[n]<top:mask>) && byte_select_match;

            // If masked bits of DBGBVR_EL1[n] are not zero, the behavior
            // is CONSTRAINED UNPREDICTABLE.
            if bvr_match && !IsZero(DBGBVR_EL1[n]<mask-1:bottom>) then
                bvr_match = ConstrainUnpredictableBool(Unpredictable_BPMASKEDBITS);
        else
            bvr_match = (vaddress<top:bottom> == DBGBVR_EL1[n]<top:bottom>) && byte_select_match;

    elsif match_cid then
        if IsInHost() then
            bvr_match = (CONTEXTIDR_EL2<31:0> == DBGBVR_EL1[n]<31:0>);
        else
            bvr_match = (PSTATE.EL IN {EL0, EL1} && CONTEXTIDR_EL1<31:0> == DBGBVR_EL1[n]<31:0>);

    elsif match_cid1 then
        bvr_match = (PSTATE.EL IN {EL0, EL1} && !IsInHost() &&
```

```
                     CONTEXTIDR_EL1<31:0> == DBGBVR_EL1[n]<31:0>);

    if match_vmid then
        bits(16) vmid;
        bits(16) bvr_vmid;

        if !Have16bitVMID() || VTCR_EL2.VS == '0' then
            vmid = ZeroExtend(VTTBR_EL2.VMID<7:0>, 16);
            bvr_vmid = ZeroExtend(DBGBVR_EL1[n]<39:32>, 16);
        else
            vmid     = VTTBR_EL2.VMID;
            bvr_vmid = DBGBVR_EL1[n]<47:32>;

        bxvr_match = (PSTATE.EL IN {EL0, EL1} && EL2Enabled() && !IsInHost() && vmid == bvr_vmid);

    elsif match_cid2 then
        bxvr_match = (PSTATE.EL != EL3 && EL2Enabled() &&
                      DBGBVR_EL1[n]<63:32> == CONTEXTIDR_EL2<31:0>);

    bvr_match_valid  = (match_addr || match_cid || match_cid1);
    bxvr_match_valid = (match_vmid || match_cid2);

    value_match = (!bxvr_match_valid || bxvr_match) && (!bvr_match_valid || bvr_match);

    return (value_match, mismatch);
```

Library pseudocode for aarch64/debug/breakpoint/AArch64.ReservedBreakpointType

```
// AArch64.ReservedBreakpointType()
// ================================
// Checks if the given DBGBCR<n>_EL1.BT2 and DBGBCR<n>_EL1.BT value is reserved and will
// generate Constrained Unpredictable behavior, otherwise returns Constraint_NONE.

(Constraint, bit, bits(4)) AArch64.ReservedBreakpointType(integer n, bit bt2_in ,bits(4) bt_in)
    bit bt2          = bt2_in;
    bits(4) bt       = bt_in;
    boolean reserved = FALSE;
    context_aware = n >= (NumBreakpointsImplemented() - NumContextAwareBreakpointsImplemented());

    if bt2 == '0' then
        // Context matching
        if !(bt IN {'0x0x'}) && !context_aware then
            reserved = TRUE;

        // EL2 extension
        if bt IN {'1xxx'} && !HaveEL(EL2) then
            reserved = TRUE;

        // Context matching
        if bt IN {'011x','11xx'} && !HaveVirtHostExt() && !HaveV82Debug() then
            reserved = TRUE;

        // Reserved
        if bt IN {'010x'} && !HaveFeatABLE() && !HaveAArch32EL(EL1) then
            reserved = TRUE;
    else
        // Reserved
        if !(bt IN {'0x0x'}) then
            reserved = TRUE;

    if reserved then
        Constraint c;
        (c, <bt2,bt>) = ConstrainUnpredictableBits(Unpredictable_RESBPTYPE, 5);
        assert c IN {Constraint_DISABLED, Constraint_UNKNOWN};
        if c == Constraint_DISABLED then
            return (c, bit UNKNOWN, bits(4) UNKNOWN);
        // Otherwise the value returned by ConstrainUnpredictableBits must be a not-reserved value

    return (Constraint_NONE, bt2, bt);
```

Library pseudocode for aarch64/debug/breakpoint/AArch64.StateMatch

```
// AArch64.StateMatch()
// ====================
// Determine whether a breakpoint or watchpoint is enabled in the current mode and state.

boolean AArch64.StateMatch(bits(2) ssc_in, bit ssce_in, bit hmc_in,
                           bits(2) pxc_in, boolean linked_in, integer linked_n_in,
                           boolean isbreakpnt, bits(64) vaddress, AccessDescriptor accdesc)
    if !HaveRME() then assert ssce_in == '0';

    // "ssc_in","ssce_in","hmc_in","pxc_in" are the control fields from
    // the DBGBCR_EL1[n] or DBGWCR_EL1[n] register.
    // "linked_in" is TRUE if this is a linked breakpoint/watchpoint type.
    // "linked_n_in" is the linked breakpoint number from the DBGBCR_EL1[n] or
    // DBGWCR_EL1[n] register.
    // "isbreakpnt" is TRUE for breakpoints, FALSE for watchpoints.
    // "vaddress" is the program counter for a linked watchpoint or the same value passed to
    // AArch64.CheckBreakpoint for a linked breakpoint.
    // "accdesc" describes the properties of the access being matched.
    bits(2) ssc      = ssc_in;
    bit ssce         = ssce_in;
    bit hmc          = hmc_in;
    bits(2) pxc      = pxc_in;
    boolean linked   = linked_in;
    integer linked_n = linked_n_in;

    // If parameters are set to a reserved type, behaves as either disabled or a defined type
    Constraint c;
    (c, ssc, ssce, hmc, pxc) = CheckValidStateMatch(ssc, ssce, hmc, pxc, isbreakpnt);
    if c == Constraint_DISABLED then return FALSE;
    // Otherwise the hmc,ssc,ssce,pxc values are either valid or the values returned by
    // CheckValidStateMatch are valid.

    EL3_match = HaveEL(EL3) && hmc == '1' && ssc<0> == '0';
    EL2_match = HaveEL(EL2) && ((hmc == '1' && (ssc:pxc != '1000')) || ssc == '11');
    EL1_match = pxc<0> == '1';
    EL0_match = pxc<1> == '1';

    boolean priv_match;
    case accdesc.el of
        when EL3  priv_match = EL3_match;
        when EL2  priv_match = EL2_match;
        when EL1  priv_match = EL1_match;
        when EL0  priv_match = EL0_match;

    // Security state match
    boolean ss_match;
    case ssce:ssc of
        when '000' ss_match = hmc == '1' || accdesc.ss != SS_Root;
        when '001' ss_match = accdesc.ss == SS_NonSecure;
        when '010' ss_match = (hmc == '1' && accdesc.ss == SS_Root) || accdesc.ss == SS_Secure;
        when '011' ss_match = (hmc == '1' && accdesc.ss != SS_Root) || accdesc.ss == SS_Secure;
        when '101' ss_match = accdesc.ss == SS_Realm;

    boolean linked_match = FALSE;
    boolean is_linked_mismatch = FALSE;

    if linked then
        // "linked_n" must be an enabled context-aware breakpoint unit. If it is not context-aware
        // then it is CONSTRAINED UNPREDICTABLE whether this gives no match, gives a match without
        // linking, or linked_n is mapped to some UNKNOWN breakpoint that is context-aware.
        if !IsContextMatchingBreakpoint(linked_n) then
            (first_ctx_cmp, last_ctx_cmp) = ContextMatchingBreakpointRange();
            (c, linked_n) = ConstrainUnpredictableInteger(first_ctx_cmp, last_ctx_cmp,
                                                          Unpredictable_BPNOTCTXCMP);
            assert c IN {Constraint_DISABLED, Constraint_NONE, Constraint_UNKNOWN};

            case c of
                when Constraint_DISABLED  return FALSE;      // Disabled
                when Constraint_NONE      linked = FALSE;    // No linking
                // Otherwise ConstrainUnpredictableInteger returned a context-aware breakpoint
```

```
    if linked then
        linked_to = TRUE;
        (linked_match, is_linked_mismatch) = AArch64.BreakpointValueMatch(linked_n, vaddress,
                                                                          linked_to, isbreakpnt);

    return (priv_match && ss_match && (!linked ||
            (!is_linked_mismatch && linked_match) || (is_linked_mismatch && !linked_match)));
```

Library pseudocode for aarch64/debug/breakpoint/DebugAddrTop

```
// DebugAddrTop()
// ==============
// Returns the value for the top bit used in Breakpoint and Watchpoint address comparisons.

integer DebugAddrTop()
    if Have56BitVAExt() then
        return 55;
    elsif Have52BitVAExt() then
        return 52;
    else
        return 48;
```

Library pseudocode for aarch64/debug/breakpoint/EffectiveMDSELR_EL1_BANK

```
// EffectiveMDSELR_EL1_BANK()
// ==========================
// Return the effective value of MDSELR_EL1.BANK.

bits(2) EffectiveMDSELR_EL1_BANK()
    // If 16 or fewer breakpoints and 16 or fewer watchpoints are implemented,
    // then the field is RES0.
    integer num_bp = NumBreakpointsImplemented();
    integer num_wp = NumWatchpointsImplemented();
    if num_bp <= 16 && num_wp <= 16 then
        return '00';

    // At EL3, the Effective value of this field is zero if MDCR_EL3.EBWE is 0.
    // At EL2, the Effective value is zero if the Effective value of MDCR_EL2.EBWE is 0.
    // That is, if either MDCR_EL3.EBWE is 0 or MDCR_EL2.EBWE is 0.
    // At EL1, the Effective value is zero if the Effective value of MDSCR_EL2.EMBWE is 0.
    // That is, if any of MDCR_EL3.EBWE, MDCR_EL2.EBWE, or MDSCR_EL1.EMBWE is 0.
    if ((HaveEL(EL3) && MDCR_EL3.EBWE == '0') ||
            (PSTATE.EL != EL3 && EL2Enabled() && MDCR_EL2.EBWE == '0') ||
            (PSTATE.EL == EL1 && MDSCR_EL1.EMBWE == '0')) then
        return '00';

    bits(2) bank = MDSELR_EL1.BANK;

    // Values are reserved depending on the number of breakpoints or watchpoints
    // implemented.
    if ((bank == '11' && num_bp <= 48 && num_wp <= 48) ||
            (bank == '10' && num_bp <= 32 && num_wp <= 32)) then
        // Reserved value
        (-, bank) = ConstrainUnpredictableBits(Unpredictable_RESMDSELR, 2);
        // The value returned by ConstrainUnpredictableBits must be a not-reserved value

    return bank;
```

Library pseudocode for aarch64/debug/breakpoint/IsBreakpointEnabled

```
// IsBreakpointEnabled()
// =====================
// Returns TRUE if the effective value of DBGBCR_EL1[n].E is '1', and FALSE otherwise.

boolean IsBreakpointEnabled(integer n)
    if (n > 15 &&
            ((!HaltOnBreakpointOrWatchpoint() && !SelfHostedExtendedBPWPEnabled()) ||
            (HaltOnBreakpointOrWatchpoint() && EDSCR2.EHBWE == '0'))) then
        return FALSE;

    return DBGBCR_EL1[n].E == '1';
```

Library pseudocode for aarch64/debug/breakpoint/SelfHostedExtendedBPWPEnabled

```
// SelfHostedExtendedBPWPEnabled()
// ===============================
// Returns TRUE if the extended breakpoints and watchpoints are enabled, and FALSE otherwise
// from a self-hosted debug perspective.

boolean SelfHostedExtendedBPWPEnabled()
    if NumBreakpointsImplemented() <= 16 && NumWatchpointsImplemented() <= 16 then
        return FALSE;

    if ((HaveEL(EL3) && MDCR_EL3.EBWE == '0') ||
            (EL2Enabled() && MDCR_EL2.EBWE == '0')) then
        return FALSE;

    return MDSCR_EL1.EMBWE == '1';
```

Library pseudocode for aarch64/debug/enables/AArch64.GenerateDebugExceptions

```
// AArch64.GenerateDebugExceptions()
// =================================

boolean AArch64.GenerateDebugExceptions()
    ss = CurrentSecurityState();
    return AArch64.GenerateDebugExceptionsFrom(PSTATE.EL, ss, PSTATE.D);
```

Library pseudocode for aarch64/debug/enables/AArch64.GenerateDebugExceptionsFrom

```
// AArch64.GenerateDebugExceptionsFrom()
// =====================================

boolean AArch64.GenerateDebugExceptionsFrom(bits(2) from_el, SecurityState from_state, bit mask)

    if OSLSR_EL1.OSLK == '1' || DoubleLockStatus() || Halted() then
        return FALSE;

    route_to_el2 = (HaveEL(EL2) && (from_state != SS_Secure || IsSecureEL2Enabled()) &&
                   (HCR_EL2.TGE == '1' || MDCR_EL2.TDE == '1'));
    target = (if route_to_el2 then EL2 else EL1);
    boolean enabled;
    if HaveEL(EL3) && from_state == SS_Secure then
        enabled = MDCR_EL3.SDD == '0';
        if from_el == EL0 && ELUsingAArch32(EL1) then
            enabled = enabled || SDER32_EL3.SUIDEN == '1';
    else
        enabled = TRUE;

    if from_el == target then
        enabled = enabled && MDSCR_EL1.KDE == '1' && mask == '0';
    else
        enabled = enabled && UInt(target) > UInt(from_el);

    return enabled;
```

Library pseudocode for aarch64/debug/ite/AArch64.TRCIT

```
// AArch64.TRCIT()
// ===============
// Determines whether an Instrumentation trace packet should
// be generated and then generates an instrumentation trace packet
// containing the value of the register passed as an argument

AArch64.TRCIT(bits(64) Xt)
    ss = CurrentSecurityState();
    if TraceInstrumentationAllowed(ss, PSTATE.EL) then
        TraceInstrumentation(Xt);
```

Library pseudocode for aarch64/debug/ite/TraceInstrumentation

```
// TraceInstrumentation()
// ======================
// Generates an instrumentation trace packet
// containing the value of the register passed as an argument

TraceInstrumentation(bits(64) Xt);
```

Library pseudocode for aarch64/debug/pmu/AArch64.ClearEventCounters

```
// AArch64.ClearEventCounters()
// ============================
// Zero all the event counters.

AArch64.ClearEventCounters()
    integer counters = AArch64.GetNumEventCountersAccessible();
    if counters != 0 then
        for idx = 0 to counters - 1
            PMEVCNTR_EL0[idx] = Zeros(64);
```

Library pseudocode for aarch64/debug/pmu/AArch64.GetNumEventCountersAccessible

```
// AArch64.GetNumEventCountersAccessible()
// ========================================
// Return the number of event counters that can be accessed at the current Exception level.

integer AArch64.GetNumEventCountersAccessible()
    integer n;
    integer total_counters = GetNumEventCounters();
    // Software can reserve some counters for EL2
    if PSTATE.EL IN {EL1, EL0} && EL2Enabled() then
        n = UInt(MDCR_EL2.HPMN);
        if n > total_counters || (!HaveFeatHPMN0() && n == 0) then
            (-, n) = ConstrainUnpredictableInteger(0, total_counters,
                                                   Unpredictable_PMUEVENTCOUNTER);
    else
        n = total_counters;

    return n;
```

Library pseudocode for aarch64/debug/pmu/AArch64.IncrementCycleCounter

```
// AArch64.IncrementCycleCounter()
// ===============================
// Increment the cycle counter and possibly set overflow bits.

AArch64.IncrementCycleCounter()
    if (CountPMUEvents(CYCLE_COUNTER_ID) &&
        (!HaveAArch32() || PMCR_EL0.LC == '1' || PMCR_EL0.D == '0' || HasElapsed64Cycles())) then
        integer old_value = UInt(PMCCNTR_EL0);
        integer new_value = old_value + 1;
        PMCCNTR_EL0 = new_value<63:0>;

        integer ovflw;
        if HaveAArch32() then
            ovflw = if PMCR_EL0.LC == '1' then 64 else 32;
        else
            ovflw = 64;

        if old_value<64:ovflw> != new_value<64:ovflw> then
            PMOVSSET_EL0.C = '1';
            PMOVSCLR_EL0.C = '1';
```

Library pseudocode for aarch64/debug/pmu/AArch64.IncrementEventCounter

```
// AArch64.IncrementEventCounter()
// ===============================
// Increment the specified event counter by the specified amount.

AArch64.IncrementEventCounter(integer idx, integer increment)
    integer old_value;
    integer new_value;
    integer ovflw;

    old_value = UInt(PMEVCNTR_EL0[idx]);
    new_value = old_value + PMUCountValue(idx, increment);

    bit lp;
    if HavePMUv3p5() then
        PMEVCNTR_EL0[idx] = new_value<63:0>;
        lp = if PMUCounterIsHyp(idx) then MDCR_EL2.HLP else PMCR_EL0.LP;
        ovflw = if lp == '1' then 64 else 32;
    else
        PMEVCNTR_EL0[idx] = ZeroExtend(new_value<31:0>, 64);
        ovflw = 32;

    if old_value<64:ovflw> != new_value<64:ovflw> then
        PMOVSSET_EL0<idx> = '1';
        PMOVSCLR_EL0<idx> = '1';
        // Check for the CHAIN event from an even counter
        if idx<0> == '0' && idx + 1 < GetNumEventCounters() && (!HavePMUv3p5() || lp == '0') then
            PMUEvent(PMU_EVENT_CHAIN, 1, idx + 1);
```

Library pseudocode for aarch64/debug/pmu/AArch64.PMUCycle

```
// AArch64.PMUCycle()
// ==================
// Called at the end of each cycle to increment event counters and
// check for PMU overflow. In pseudocode, a cycle ends after the
// execution of the operational pseudocode.

AArch64.PMUCycle()
    if !HavePMUv3() then
        return;

    PMUEvent(PMU_EVENT_CPU_CYCLES);

    integer counters = GetNumEventCounters();
    if counters != 0 then
        for idx = 0 to counters - 1
            if CountPMUEvents(idx) then
                integer accumulated = PMUEventAccumulator[idx];
                AArch64.IncrementEventCounter(idx, accumulated);
            PMUEventAccumulator[idx] = 0;

    AArch64.IncrementCycleCounter();
    CheckForPMUOverflow();
```

Library pseudocode for aarch64/debug/pmu/AArch64.PMUSwIncrement

```
// AArch64.PMUSwIncrement()
// ========================
// Generate PMU Events on a write to PMSWINC_EL0.

AArch64.PMUSwIncrement(bits(32) sw_incr)
    integer counters = AArch64.GetNumEventCountersAccessible();
    if counters != 0 then
        for idx = 0 to counters - 1
            if sw_incr<idx> == '1' then
                PMUEvent(PMU_EVENT_SW_INCR, 1, idx);
```

Library pseudocode for aarch64/debug/statisticalprofiling/CollectContextIDR1

```
// CollectContextIDR1()
// ====================

boolean CollectContextIDR1()
    if !StatisticalProfilingEnabled() then return FALSE;
    if PSTATE.EL == EL2 then return FALSE;
    if EL2Enabled() && HCR_EL2.TGE == '1' then return FALSE;
    return PMSCR_EL1.CX == '1';
```

Library pseudocode for aarch64/debug/statisticalprofiling/CollectContextIDR2

```
// CollectContextIDR2()
// ====================

boolean CollectContextIDR2()
    if !StatisticalProfilingEnabled() then return FALSE;
    if !EL2Enabled() then return FALSE;
    return PMSCR_EL2.CX == '1';
```

Library pseudocode for aarch64/debug/statisticalprofiling/CollectPhysicalAddress

```
// CollectPhysicalAddress()
// ========================

boolean CollectPhysicalAddress()
    if !StatisticalProfilingEnabled() then return FALSE;
    (owning_ss, owning_el) = ProfilingBufferOwner();
    if HaveEL(EL2) && (owning_ss != SS_Secure || IsSecureEL2Enabled()) then
        return PMSCR_EL2.PA == '1' && (owning_el == EL2 || PMSCR_EL1.PA == '1');
    else
        return PMSCR_EL1.PA == '1';
```

Library pseudocode for aarch64/debug/statisticalprofiling/CollectTimeStamp

```
// CollectTimeStamp()
// ==================

TimeStamp CollectTimeStamp()
    if !StatisticalProfilingEnabled() then return TimeStamp_None;
    (-, owning_el) = ProfilingBufferOwner();

    if owning_el == EL2 then
        if PMSCR_EL2.TS == '0' then return TimeStamp_None;
    else
        if PMSCR_EL1.TS == '0' then return TimeStamp_None;

    bits(2) PCT_el1;
    if !HaveECVExt() then
        PCT_el1 = '0':PMSCR_EL1.PCT<0>;       // PCT<1> is RES0
    else
        PCT_el1 = PMSCR_EL1.PCT;
        if PCT_el1 == '10' then
            // Reserved value
            (-, PCT_el1) = ConstrainUnpredictableBits(Unpredictable_PMSCR_PCT, 2);
    if EL2Enabled() then
        bits(2) PCT_el2;
        if !HaveECVExt() then
            PCT_el2 = '0':PMSCR_EL2.PCT<0>;   // PCT<1> is RES0
        else
            PCT_el2 = PMSCR_EL2.PCT;
            if PCT_el2 == '10' then
                // Reserved value
                (-, PCT_el2) = ConstrainUnpredictableBits(Unpredictable_PMSCR_PCT, 2);
        case PCT_el2 of
            when '00'
                return if IsInHost() then TimeStamp_Physical else TimeStamp_Virtual;
            when '01'
                if owning_el == EL2 then return TimeStamp_Physical;
            when '11'
                assert HaveECVExt();          // FEAT_ECV must be implemented
                if owning_el == EL1 && PCT_el1 == '00' then
                    return if IsInHost() then TimeStamp_Physical else TimeStamp_Virtual;
                else
                    return TimeStamp_OffsetPhysical;
            otherwise
                Unreachable();

    case PCT_el1 of
        when '00' return if IsInHost() then TimeStamp_Physical else TimeStamp_Virtual;
        when '01' return TimeStamp_Physical;
        when '11'
            assert HaveECVExt();              // FEAT_ECV must be implemented
            return TimeStamp_OffsetPhysical;
        otherwise Unreachable();
```

Library pseudocode for aarch64/debug/statisticalprofiling/OpType

```
// OpType
// ======
// Types of operation filtered by SPECollectRecord().

enumeration OpType {
    OpType_Load,        // Any memory-read operation other than atomics, compare-and-swap, and swap
    OpType_Store,       // Any memory-write operation, including atomics without return
    OpType_LoadAtomic,  // Atomics with return, compare-and-swap and swap
    OpType_Branch,      // Software write to the PC
    OpType_Other        // Any other class of operation
};
```

Library pseudocode for aarch64/debug/statisticalprofiling/ProfilingBufferEnabled

```
// ProfilingBufferEnabled()
// ========================

boolean ProfilingBufferEnabled()
    if !HaveStatisticalProfiling() then return FALSE;
    (owning_ss, owning_el) = ProfilingBufferOwner();
    bits(2) state_bits;
    if HaveRME() then
        state_bits = SCR_EL3.NSE : EffectiveSCR_EL3_NS();
    else
        state_bits = '0' : SCR_EL3.NS;

    boolean state_match;
    case owning_ss of
        when SS_Secure    state_match = state_bits == '00';
        when SS_NonSecure state_match = state_bits == '01';
        when SS_Realm     state_match = state_bits == '11';
    return (!ELUsingAArch32(owning_el) && state_match &&
            PMBLIMITR_EL1.E == '1' && PMBSR_EL1.S == '0');
```

Library pseudocode for aarch64/debug/statisticalprofiling/ProfilingBufferOwner

```
// ProfilingBufferOwner()
// ======================

(SecurityState, bits(2)) ProfilingBufferOwner()
    SecurityState owning_ss;

    if HaveEL(EL3) then
        bits(3) state_bits;
        if HaveRME() then
            state_bits = MDCR_EL3.<NSPBE,NSPB>;
            if (state_bits IN {'10x'} ||
                (!HaveSecureEL2Ext() && state_bits IN {'00x'})) then
                // Reserved value
                (-, state_bits) = ConstrainUnpredictableBits(Unpredictable_RESERVEDNSxB, 3);
        else
            state_bits = '0' : MDCR_EL3.NSPB;

        case state_bits of
            when '00x' owning_ss = SS_Secure;
            when '01x' owning_ss = SS_NonSecure;
            when '11x' owning_ss = SS_Realm;
    else
        owning_ss = if SecureOnlyImplementation() then SS_Secure else SS_NonSecure;

    bits(2) owning_el;
    if HaveEL(EL2) && (owning_ss != SS_Secure || IsSecureEL2Enabled()) then
        owning_el = if MDCR_EL2.E2PB == '00' then EL2 else EL1;
    else
        owning_el = EL1;

    return (owning_ss, owning_el);
```

Library pseudocode for aarch64/debug/statisticalprofiling/ProfilingSynchronizationBarrier

```
// ProfilingSynchronizationBarrier()
// =================================
// Barrier to ensure that all existing profiling data has been formatted, and profiling buffer
// addresses have been translated such that writes to the profiling buffer have been initiated.
// A following DSB completes when writes to the profiling buffer have completed.

ProfilingSynchronizationBarrier();
```

Library pseudocode for aarch64/debug/statisticalprofiling/SPEAddByteToRecord

```
// SPEAddByteToRecord()
// ====================
// Add one byte to a record and increase size property appropriately.

SPEAddByteToRecord(bits(8) b)
    assert SPERecordSize < SPEMaxRecordSize;
    SPERecordData[SPERecordSize] = b;
    SPERecordSize = SPERecordSize + 1;
```

Library pseudocode for aarch64/debug/statisticalprofiling/SPEAddPacketToRecord

```
// SPEAddPacketToRecord()
// ======================
// Add passed header and payload data to the record.
// Payload must be a multiple of 8.

SPEAddPacketToRecord(bits(2) header_hi, bits(4) header_lo,
                        bits(N) payload)
    assert N MOD 8 == 0;
    bits(2) sz;
    case N of
        when 8 sz = '00';
        when 16 sz = '01';
        when 32 sz = '10';
        when 64 sz = '11';
        otherwise Unreachable();

    bits(8) header = header_hi:sz:header_lo;
    SPEAddByteToRecord(header);
    for i = 0 to (N DIV 8)-1
        SPEAddByteToRecord(payload<i*8+7:i*8>);
```

Library pseudocode for aarch64/debug/statisticalprofiling/SPEBranch

```
// SPEBranch()
// ===========
// Called on every branch if SPE is present. Maintains previous branch target
// and branch related SPE functionality.

SPEBranch(bits(N) target, BranchType branch_type, boolean conditional, boolean taken_flag)
    boolean is_isb = FALSE;
    SPEBranch(target, branch_type, conditional, taken_flag, is_isb);

SPEBranch(bits(N) target, BranchType branch_type, boolean conditional, boolean taken_flag,
          boolean is_isb)
    // If the PE implements branch prediction, data about (mis)prediction is collected
    // through the PMU events.

    boolean collect_prev_br;
    boolean collect_prev_br_eret = boolean IMPLEMENTATION_DEFINED "SPE prev br on eret";
    boolean collect_prev_br_exception = boolean IMPLEMENTATION_DEFINED "SPE prev br on exception";
    boolean collect_prev_br_isb = boolean IMPLEMENTATION_DEFINED "SPE prev br on isb";
    case branch_type of
        when BranchType_EXCEPTION
            collect_prev_br = collect_prev_br_exception;
        when BranchType_ERET
            collect_prev_br = collect_prev_br_eret;
        otherwise
            collect_prev_br = !is_isb || collect_prev_br_isb;

    // Implements previous branch target functionality
    if (taken_flag && !IsZero(PMSIDR_EL1.PBT) && StatisticalProfilingEnabled() &&
            collect_prev_br) then

        if SPESampleInFlight then
            // Save the target address for it to be added to record.
            bits(64) previous_target = SPESamplePreviousBranchAddress;
            SPESampleAddress[SPEAddrPosPrevBranchTarget]<63:0> = previous_target<63:0>;
            boolean previous_branch_valid = SPESamplePreviousBranchAddressValid;
            SPESampleAddressValid[SPEAddrPosPrevBranchTarget] = previous_branch_valid;
        SPESamplePreviousBranchAddress<55:0> = target<55:0>;

        bit ns;
        bit nse;
        case CurrentSecurityState() of
            when SS_Secure
                ns = '0';
                nse = '0';
            when SS_NonSecure
                ns = '1';
                nse = '0';
            when SS_Realm
                ns = '1';
                nse = '1';
            otherwise Unreachable();

        SPESamplePreviousBranchAddress<63> = ns;
        SPESamplePreviousBranchAddress<60> = nse;
        SPESamplePreviousBranchAddress<62:61> = PSTATE.EL;
        SPESamplePreviousBranchAddressValid = TRUE;

    if !StatisticalProfilingEnabled() then
        if taken_flag then
            // Invalidate previous branch address, if profiling is disabled
            // or prohibited.
            SPESamplePreviousBranchAddressValid = FALSE;
        return;

    if SPESampleInFlight then
        is_direct = branch_type IN {BranchType_DIR, BranchType_DIRCALL};
        SPESampleClass = '10';
        SPESampleSubclass<1> = if is_direct then '0' else '1';
        SPESampleSubclass<0> = if conditional then '1' else '0';
        SPESampleOpType = OpType_Branch;
```

```
        // Save the target address.
        if taken_flag then
            SPESampleAddress[SPEAddrPosBranchTarget]<55:0> = target<55:0>;

            bit ns;
            bit nse;
            case CurrentSecurityState() of
                when SS_Secure
                    ns = '0';
                    nse = '0';
                when SS_NonSecure
                    ns = '1';
                    nse = '0';
                when SS_Realm
                    ns = '1';
                    nse = '1';
                otherwise Unreachable();

            SPESampleAddress[SPEAddrPosBranchTarget]<63> = ns;
            SPESampleAddress[SPEAddrPosBranchTarget]<60> = nse;
            SPESampleAddress[SPEAddrPosBranchTarget]<62:61> = PSTATE.EL;
            SPESampleAddressValid[SPEAddrPosBranchTarget] = TRUE;

        SPESampleEvents<6> = if !taken_flag then '1' else '0';
```

Library pseudocode for aarch64/debug/statisticalprofiling/SPEBufferFilled

```
// SPEBufferFilled()
// =================
// Deal with a full buffer event.

SPEBufferFilled()
    if IsZero(PMBSR_EL1.S) then
        PMBSR_EL1.S = '1';                          // Assert PMBIRQ
        PMBSR_EL1.EC = '000000';                    // Other buffer management event
        PMBSR_EL1.MSS = ZeroExtend('000001', 16);   // Set buffer full event

    PMUEvent(PMU_EVENT_SAMPLE_WRAP);
```

Library pseudocode for aarch64/debug/statisticalprofiling/SPEBufferIsFull

```
// SPEBufferIsFull()
// =================
// Return true if another full size sample record would not fit in the
// profiling buffer.

boolean SPEBufferIsFull()
    integer write_pointer_limit = UInt(PMBLIMITR_EL1.LIMIT:Zeros(12));
    integer current_write_pointer = UInt(PMBPTR_EL1);
    integer record_max_size = 1<<UInt(PMSIDR_EL1.MaxSize);
    return current_write_pointer > (write_pointer_limit - record_max_size);
```

Library pseudocode for aarch64/debug/statisticalprofiling/SPECollectRecord

```
// SPECollectRecord()
// ==================
// Returns TRUE if the sampled class of instructions or operations, as
// determined by PMSFCR_EL1, are recorded and FALSE otherwise.

boolean SPECollectRecord(bits(64) events, integer total_latency, OpType optype)
    assert StatisticalProfilingEnabled();

    bits(64) mask = 0xAA<63:0>;                              // Bits [7,5,3,1]
    bits(64) e;
    bits(64) m;
    if HaveSVE() then mask<18:17> = '11';                    // Predicate flags
    if HaveTME() then mask<16> = '1';
    if HaveStatisticalProfilingv1p1() then mask<11> = '1';  // Alignment Flag
    if HaveStatisticalProfilingv1p2() then mask<6> = '1';   // Not taken flag
    if HaveStatisticalProfilingv1p4() then
        mask<10:8,4,2> = '11111';
    else
        bits(5) impdef_mask;
        impdef_mask = bits(5) IMPLEMENTATION_DEFINED "SPE mask 10:8,4,2";
        mask<10:8,4,2> = impdef_mask;

    mask<63:48> = bits(16) IMPLEMENTATION_DEFINED "SPE mask 63:48";
    mask<31:24> = bits(8) IMPLEMENTATION_DEFINED "SPE mask 31:24";
    mask<15:12> = bits(4) IMPLEMENTATION_DEFINED "SPE mask 15:12";

    e = events AND mask;
    boolean is_rejected_nevent = FALSE;
    boolean is_nevt;
    // Filtering by inverse event
    if HaveStatisticalProfilingv1p2() then
        m = PMSNEVFR_EL1 AND mask;
        is_nevt = IsZero(e AND m);
        if PMSFCR_EL1.FnE == '1' then
            // Inverse filtering by event is enabled
            if !IsZero(m) then
                // Not UNPREDICTABLE case
                is_rejected_nevent = !is_nevt;
            else
                is_rejected_nevent = ConstrainUnpredictableBool(Unpredictable_BADPMSFCR);
    else
        is_nevt = TRUE; // not implemented

    boolean is_rejected_event = FALSE;

    // Filtering by event
    m = PMSEVFR_EL1 AND mask;
    boolean is_evt = IsZero(NOT(e) AND m);
    if PMSFCR_EL1.FE == '1' then
        // Filtering by event is enabled
        if !IsZero(m) then
            // Not UNPREDICTABLE case
            is_rejected_event = !is_evt;
        else
            is_rejected_event = ConstrainUnpredictableBool(Unpredictable_BADPMSFCR);

    if (HaveStatisticalProfilingv1p2() && PMSFCR_EL1.<FnE,FE> == '11' &&
            !IsZero(PMSEVFR_EL1 AND PMSNEVFR_EL1 AND mask)) then
        // UNPREDICTABLE case due to combination of filter and inverse filter
        is_rejected_nevent = ConstrainUnpredictableBool(Unpredictable_BADPMSFCR);
        is_rejected_event = ConstrainUnpredictableBool(Unpredictable_BADPMSFCR);

    if is_evt && is_nevt then
        PMUEvent(PMU_EVENT_SAMPLE_FEED_EVENT);

    boolean is_op_br = FALSE;
    boolean is_op_ld = FALSE;
    boolean is_op_st = FALSE;

    is_op_br = (optype == OpType_Branch);
```

```
    is_op_ld = (optype IN {OpType_Load, OpType_LoadAtomic});
    is_op_st = (optype IN {OpType_Store, OpType_LoadAtomic});

    if is_op_br then PMUEvent(PMU_EVENT_SAMPLE_FEED_BR);
    if is_op_ld then PMUEvent(PMU_EVENT_SAMPLE_FEED_LD);
    if is_op_st then PMUEvent(PMU_EVENT_SAMPLE_FEED_ST);

    boolean is_op = ((is_op_br && PMSFCR_EL1.B == '1') ||
                     (is_op_ld && PMSFCR_EL1.LD == '1') ||
                     (is_op_st && PMSFCR_EL1.ST == '1'));

    if is_op then PMUEvent(PMU_EVENT_SAMPLE_FEED_OP);

    // Filter by type
    boolean is_rejected_type = FALSE;
    if PMSFCR_EL1.FT == '1' then
        // Filtering by type is enabled
        if !IsZero(PMSFCR_EL1.<B, LD, ST>) then
            // Not an UNPREDICTABLE case
            is_rejected_type = !is_op;
        else
            is_rejected_type = ConstrainUnpredictableBool(Unpredictable_BADPMSFCR);

    // Filter by latency
    boolean is_rejected_latency = FALSE;
    boolean is_lat = (total_latency < UInt(PMSLATFR_EL1.MINLAT));
    if is_lat then PMUEvent(PMU_EVENT_SAMPLE_FEED_LAT);

    if PMSFCR_EL1.FL == '1' then
        // Filtering by latency is enabled
        if !IsZero(PMSLATFR_EL1.MINLAT) then
            // Not an UNPREDICTABLE case
            is_rejected_latency = !is_lat;
        else
            is_rejected_latency = ConstrainUnpredictableBool(Unpredictable_BADPMSFCR);

    boolean is_rejected_data_source;
    // Filtering by Data Source
    if (HaveStatisticalProfilingFDS() && PMSFCR_EL1.FDS == '1' &&
            is_op_ld && SPESampleDataSourceValid) then
        bits(16) data_source = SPESampleDataSource;
        integer index = UInt(data_source<5:0>);
        is_rejected_data_source = PMSDSFR_EL1<index> == '0';
    else
        is_rejected_data_source = FALSE;

    boolean return_value;
    return_value = !(is_rejected_nevent || is_rejected_event ||
                      is_rejected_type || is_rejected_latency);

    if return_value then
        PMUEvent(PMU_EVENT_SAMPLE_FILTRATE);
    return return_value;
```

Library pseudocode for aarch64/debug/statisticalprofiling/SPEConstructRecord

```
// SPEConstructRecord()
// ====================
// Create new record and populate it with packets using sample storage data.
// This is an example implementation, packets may appear in
// any order as long as the record ends with an End or Timestamp packet.

SPEConstructRecord()
    integer payload_size;

    // Empty the record.
    SPEEmptyRecord();

    // Add contextEL1 if available
    if SPESampleContextEL1Valid then
        SPEAddPacketToRecord('01', '0100', SPESampleContextEL1);

     // Add contextEL2 if available
    if SPESampleContextEL2Valid then
        SPEAddPacketToRecord('01', '0101', SPESampleContextEL2);

    // Add valid counters
    for counter_index = 0 to (SPEMaxCounters - 1)
        if SPESampleCounterValid[counter_index] then
            if counter_index >= 8 then
                // Need extended format
                SPEAddByteToRecord('001000':counter_index<4:3>);
            // Check for overflow
            boolean large_counters = boolean IMPLEMENTATION_DEFINED "SPE 16bit counters";
            if SPESampleCounter[counter_index] > 0xFFFF && large_counters then
                SPESampleCounter[counter_index] = 0xFFFF;
            elsif SPESampleCounter[counter_index] > 0xFFF then
                SPESampleCounter[counter_index] = 0xFFF;

            // Add byte0 for short format (byte1 for extended format)
            SPEAddPacketToRecord('10', '1':counter_index<2:0>,
                SPESampleCounter[counter_index]<15:0>);

    // Add valid addresses
    if HaveStatisticalProfilingv1p2() then
        // Under the some conditions, it is IMPLEMENTATION_DEFINED whether
        // previous branch packet is present.
        boolean include_prev_br = boolean IMPLEMENTATION_DEFINED "SPE get prev br if not br";
        if SPESampleOpType != OpType_Branch && !include_prev_br then
            SPESampleAddressValid[SPEAddrPosPrevBranchTarget] = FALSE;

    // Data Virtual address should not be collected if this was an NV2 access and Statistical
    // Profiling is disabled at EL2.
    if !StatisticalProfilingEnabled(EL2) && SPESampleInstIsNV2 then
        SPESampleAddressValid[SPEAddrPosDataVirtual] = FALSE;

    for address_index = 0 to (SPEMaxAddrs - 1)
        if SPESampleAddressValid[address_index] then
            if address_index >= 8 then
                // Need extended format
                SPEAddByteToRecord('001000':address_index<4:3>);
            // Add byte0 for short format (byte1 for extended format)
            SPEAddPacketToRecord('10', '0':address_index<2:0>,
                SPESampleAddress[address_index]);

    // Add Data Source
    if SPESampleDataSourceValid then
        payload_size = SPEGetDataSourcePayloadSize();
        SPEAddPacketToRecord('01', '0011', SPESampleDataSource<8*payload_size-1:0>);

    // Add operation details
    SPEAddPacketToRecord('01', '10':SPESampleClass, SPESampleSubclass);

    // Add events
    // Get size of payload in bytes.
    payload_size = SPEGetEventsPayloadSize();
```

```
    SPEAddPacketToRecord('01', '0010', SPESampleEvents<8*payload_size-1:0>);

    // Add Timestamp to end the record if one is available.
    // Otherwise end with an End packet.
    if SPESampleTimestampValid then
        SPEAddPacketToRecord('01', '0001', SPESampleTimestamp);
    else
        SPEAddByteToRecord('00000001');

    // Add padding
    while SPERecordSize MOD (1<<UInt(PMBIDR_EL1.Align)) != 0 do
        SPEAddByteToRecord(Zeros(8));
    SPEWriteToBuffer();
```

Library pseudocode for aarch64/debug/statisticalprofiling/SPECycle

```
// SPECycle()
// ==========
// Function called at the end of every cycle. Responsible for asserting interrupts
// and advancing counters.

SPECycle()
    if !HaveStatisticalProfiling() then
        return;

    // Increment pending counters
    if SPESampleInFlight then
        for i = 0 to (SPEMaxCounters - 1)
            if SPESampleCounterPending[i] then
                SPESampleCounter[i] = SPESampleCounter[i] + 1;

    // Assert PMBIRQ if appropriate.
    SetInterruptRequestLevel(InterruptID_PMBIRQ,
                             if PMBSR_EL1.S == '1' then Signal_High else Signal_Low);
```

Library pseudocode for aarch64/debug/statisticalprofiling/SPEEmptyRecord

```
// SPEEmptyRecord()
// ================
// Reset record data.

SPEEmptyRecord()
    SPERecordSize = 0;
    for i = 0 to (SPEMaxRecordSize - 1)
        SPERecordData[i] = Zeros(8);
```

Library pseudocode for aarch64/debug/statisticalprofiling/SPEEvent

```
// SPEEvent()
// ==========
// Called by PMUEvent if a sample is in flight.
// Sets appropriate bit in SPESampleStorage.events.

SPEEvent(bits(16) pmuevent)
    case pmuevent of
        when PMU_EVENT_DSNP_HIT_RD
            if HaveStatisticalProfilingv1p4() then
                SPESampleEvents<23> = '1';
        when PMU_EVENT_L1D_LFB_HIT_RD
            if HaveStatisticalProfilingv1p4() then
                SPESampleEvents<22> = '1';
        when PMU_EVENT_L2D_LFB_HIT_RD
            if HaveStatisticalProfilingv1p4() then
                SPESampleEvents<22> = '1';
        when PMU_EVENT_L3D_LFB_HIT_RD
            if HaveStatisticalProfilingv1p4() then
                SPESampleEvents<22> = '1';
        when PMU_EVENT_LL_LFB_HIT_RD
            if HaveStatisticalProfilingv1p4() then
                SPESampleEvents<22> = '1';
        when PMU_EVENT_L1D_CACHE_HITM_RD
            if HaveStatisticalProfilingv1p4() then
                SPESampleEvents<21> = '1';
        when PMU_EVENT_L2D_CACHE_HITM_RD
            if HaveStatisticalProfilingv1p4() then
                SPESampleEvents<21> = '1';
        when PMU_EVENT_L3D_CACHE_HITM_RD
            if HaveStatisticalProfilingv1p4() then
                SPESampleEvents<21> = '1';
        when PMU_EVENT_LL_CACHE_HITM_RD
            if HaveStatisticalProfilingv1p4() then
                SPESampleEvents<21> = '1';
        when PMU_EVENT_L2D_CACHE_LMISS_RD
            if HaveStatisticalProfilingv1p4() then
                SPESampleEvents<20> = '1';
        when PMU_EVENT_L2D_CACHE_RD
            if HaveStatisticalProfilingv1p4() then
                SPESampleEvents<19> = '1';
        when PMU_EVENT_SVE_PRED_EMPTY_SPEC
            if HaveStatisticalProfilingv1p1() then
                SPESampleEvents<18> = '1';
        when PMU_EVENT_SVE_PRED_PARTIAL_SPEC
            if HaveStatisticalProfilingv1p1() then
                SPESampleEvents<17> = '1';
        when PMU_EVENT_LDST_ALIGN_LAT
            if HaveStatisticalProfilingv1p1() then
                SPESampleEvents<11> = '1';
        when PMU_EVENT_REMOTE_ACCESS         SPESampleEvents<10>  = '1';
        when PMU_EVENT_LL_CACHE_MISS         SPESampleEvents<9>   = '1';
        when PMU_EVENT_LL_CACHE              SPESampleEvents<8>   = '1';
        when PMU_EVENT_BR_MIS_PRED           SPESampleEvents<7>   = '1';
        when PMU_EVENT_BR_MIS_PRED_RETIRED   SPESampleEvents<7>   = '1';
        when PMU_EVENT_DTLB_WALK             SPESampleEvents<5>   = '1';
        when PMU_EVENT_L1D_TLB               SPESampleEvents<4>   = '1';
        when PMU_EVENT_L1D_CACHE_REFILL
            if !HaveStatisticalProfilingv1p4() then
                SPESampleEvents<3> = '1';
        when PMU_EVENT_L1D_CACHE_LMISS_RD
            if HaveStatisticalProfilingv1p4() then
                SPESampleEvents<3> = '1';
        when PMU_EVENT_L1D_CACHE             SPESampleEvents<2>   = '1';
        when PMU_EVENT_INST_RETIRED          SPESampleEvents<1>   = '1';
        when PMU_EVENT_EXC_TAKEN             SPESampleEvents<0>   = '1';
        otherwise return;
    return;
```

Library pseudocode for aarch64/debug/statisticalprofiling/SPEGetDataSourcePayloadSize

```
// SPEGetDataSourcePayloadSize()
// =============================
// Returns the size of the Data Source payload in bytes.

integer SPEGetDataSourcePayloadSize()
    return integer IMPLEMENTATION_DEFINED "SPE Data Source packet payload size";
```

Library pseudocode for aarch64/debug/statisticalprofiling/SPEGetEventsPayloadSize

```
// SPEGetEventsPayloadSize()
// =========================
// Returns the size in bytes of the Events packet payload as an integer.

integer SPEGetEventsPayloadSize()
    integer size = integer IMPLEMENTATION_DEFINED "SPE Events packet payload size";
    return size;
```

Library pseudocode for aarch64/debug/statisticalprofiling/SPEGetRandomBoolean

```
// SPEGetRandomBoolean()
// =====================
// Returns a random or pseudo-random boolean value.

boolean SPEGetRandomBoolean();
```

Library pseudocode for aarch64/debug/statisticalprofiling/SPEGetRandomInterval

```
// SPEGetRandomInterval()
// ======================
// Returns a random or pseudo-random byte for resetting COUNT or ECOUNT.

bits(8) SPEGetRandomInterval();
```

Library pseudocode for aarch64/debug/statisticalprofiling/SPEISB

```
// SPEISB()
// ========
// Called by ISB instruction, correctly calls SPEBranch to save previous branches.

SPEISB()
    bits(64) address = PC[] + 4;
    BranchType branch_type = BranchType_DIR;
    boolean branch_conditional = FALSE;
    boolean taken = FALSE;
    boolean is_isb = TRUE;

    SPEBranch(address, branch_type, branch_conditional, taken, is_isb);
```

Library pseudocode for aarch64/debug/statisticalprofiling/SPEMaxAddrs

```
constant integer SPEMaxAddrs = 32;
```

Library pseudocode for aarch64/debug/statisticalprofiling/SPEMaxCounters

```
constant integer SPEMaxCounters = 32;
```

Library pseudocode for aarch64/debug/statisticalprofiling/SPEMaxRecordSize

```
constant integer SPEMaxRecordSize = 64;
```

Library pseudocode for aarch64/debug/statisticalprofiling/SPEPostExecution

```
constant integer SPEAddrPosPCVirtual = 0;
constant integer SPEAddrPosBranchTarget = 1;
constant integer SPEAddrPosDataVirtual = 2;
constant integer SPEAddrPosDataPhysical = 3;
constant integer SPEAddrPosPrevBranchTarget = 4;
constant integer SPECounterPosTotalLatency = 0;
constant integer SPECounterPosIssueLatency = 1;
constant integer SPECounterPosTranslationLatency = 2;
boolean SPESampleInFlight = FALSE;
bits(32) SPESampleContextEL1;
boolean SPESampleContextEL1Valid;
bits(32) SPESampleContextEL2;
boolean SPESampleContextEL2Valid;
boolean SPESampleInstIsNV2 = FALSE;
bits(64) SPESamplePreviousBranchAddress;
boolean SPESamplePreviousBranchAddressValid;
bits(16) SPESampleDataSource;
boolean SPESampleDataSourceValid;
OpType SPESampleOpType;
bits(2) SPESampleClass;
bits(8) SPESampleSubclass;
boolean SPESampleSubclassValid;
bits(64) SPESampleTimestamp;
boolean SPESampleTimestampValid;
bits(64) SPESampleEvents;

// SPEPostExecution()
// ==================
// Called after every executed instruction.

SPEPostExecution()
    if SPESampleInFlight then
        SPESampleInFlight = FALSE;
        PMUEvent(PMU_EVENT_SAMPLE_FEED);

        // Stop any pending counters
        for counter_index = 0 to (SPEMaxCounters - 1)
            if SPESampleCounterPending[counter_index] then
                SPEStopCounter(counter_index);

        boolean discard = FALSE;
        if HaveStatisticalProfilingv1p2() then
            discard = PMBLIMITR_EL1.FM == '10';
        if SPECollectRecord(SPESampleEvents,
                            SPESampleCounter[SPECounterPosTotalLatency],
                            SPESampleOpType) && !discard then
            SPEConstructRecord();
            if SPEBufferIsFull() then
                SPEBufferFilled();

        SPEResetSampleStorage();

// Counter storage
array [0..SPEMaxCounters-1] of integer SPESampleCounter;

array [0..SPEMaxCounters-1] of boolean SPESampleCounterValid;

array [0..SPEMaxCounters-1] of boolean SPESampleCounterPending;

// Address storage
array [0..SPEMaxAddrs-1] of bits(64) SPESampleAddress;

array [0..SPEMaxAddrs-1] of boolean SPESampleAddressValid;
```

Library pseudocode for aarch64/debug/statisticalprofiling/SPEPreExecution

```
// SPEPreExecution()
// =================
// Called prior to execution, for all instructions.

SPEPreExecution()
    if StatisticalProfilingEnabled() then
        PMUEvent(PMU_EVENT_SAMPLE_POP);
        if SPEToCollectSample() then
            if !SPESampleInFlight then
                SPESampleInFlight = TRUE;

                // Start total latency and issue latency counters for SPE
                SPEStartCounter(SPECounterPosTotalLatency);
                SPEStartCounter(SPECounterPosIssueLatency);

                SPESampleAddContext();

                SPESampleAddAddressPCVirtual();

                // Timestamp may be collected at any point in the sampling operation.
                // Collecting prior to execution is one possible choice.
                // This choice is IMPLEMENTATION_DEFINED.
                SPESampleAddTimeStamp();
            else
                PMUEvent(PMU_EVENT_SAMPLE_COLLISION);
                PMBSR_EL1.COLL = '1';

            // Many operations are type other and not conditional, can save footprint
            // and overhead by having this as the default and not calling SPESampleAddOpOther
            // if conditional == FALSE
            SPESampleAddOpOther(FALSE);
```

Library pseudocode for aarch64/debug/statisticalprofiling/SPEResetSampleCounter

```
// SPEResetSampleCounter()
// =======================
// Reset PMSICR_EL1.Counter

SPEResetSampleCounter()
    PMSICR_EL1.COUNT<31:8> = PMSIRR_EL1.INTERVAL;
    if PMSIRR_EL1.RND == '1' && PMSIDR_EL1.ERnd == '0' then
        PMSICR_EL1.COUNT<7:0> = SPEGetRandomInterval();
    else
        PMSICR_EL1.COUNT<7:0> = Zeros(8);
```

Library pseudocode for aarch64/debug/statisticalprofiling/SPEResetSampleStorage

```
integer SPERecordSize;

// SPEResetSampleStorage()
// =======================
// Reset all variables inside sample storage.

SPEResetSampleStorage()
    // Context values
    SPESampleContextEL1 = Zeros(32);
    SPESampleContextEL1Valid = FALSE;
    SPESampleContextEL2 = Zeros(32);
    SPESampleContextEL2Valid = FALSE;

    // Counter values
    for i = 0 to (SPEMaxCounters - 1)
        SPESampleCounter[i] = 0;
        SPESampleCounterValid[i] = FALSE;
        SPESampleCounterPending[i] = FALSE;

    // Address values
    for i = 0 to (SPEMaxAddrs - 1)
        SPESampleAddressValid[i] = FALSE;
        SPESampleAddress[i] = Zeros(64);

    // Data source values
    SPESampleDataSource = Zeros(16);
    SPESampleDataSourceValid = FALSE;

    // Operation values
    SPESampleClass = Zeros(2);
    SPESampleSubclass = Zeros(8);
    SPESampleSubclassValid = FALSE;

    // Timestamp values
    SPESampleTimestamp = Zeros(64);
    SPESampleTimestampValid = FALSE;

    // Event values
    SPESampleEvents<63:48> = bits(16) IMPLEMENTATION_DEFINED "SPE EVENTS 63_48";
    SPESampleEvents<47:32> = Zeros(16);
    SPESampleEvents<31:24> = bits(8) IMPLEMENTATION_DEFINED "SPE EVENTS 31_24";
    SPESampleEvents<23:16> = Zeros(8);
    SPESampleEvents<15:12> = bits(4) IMPLEMENTATION_DEFINED "SPE EVENTS 15_12";
    SPESampleEvents<11:0> = Zeros(12);

    SPESampleInstIsNV2 = FALSE;

array [0..SPEMaxRecordSize-1] of bits(8) SPERecordData;
```

Library pseudocode for aarch64/debug/statisticalprofiling/SPESampleAddAddressPCVirtual

```
// SPESampleAddAddressPCVirtual()
// ==============================
// Save the current PC address to sample storage.

SPESampleAddAddressPCVirtual()
    bits(64) this_address = ThisInstrAddr(64);
    SPESampleAddress[SPEAddrPosPCVirtual]<55:0> = this_address<55:0>;

    bit ns;
    bit nse;
    case CurrentSecurityState() of
        when SS_Secure
            ns = '0';
            nse = '0';
        when SS_NonSecure
            ns = '1';
            nse = '0';
        when SS_Realm
            ns = '1';
            nse = '1';
        otherwise Unreachable();

    bits(2) el = PSTATE.EL;
    SPESampleAddress[SPEAddrPosPCVirtual]<63:56> = ns:el:nse:Zeros(4);
    SPESampleAddressValid[SPEAddrPosPCVirtual] = TRUE;
```

Library pseudocode for aarch64/debug/statisticalprofiling/SPESampleAddContext

```
// SPESampleAddContext()
// =====================
// Save contexts to sample storage if appropriate.

SPESampleAddContext()
    if CollectContextIDR1() then
        SPESampleContextEL1 = CONTEXTIDR_EL1<31:0>;
        SPESampleContextEL1Valid = TRUE;
    if CollectContextIDR2() then
        SPESampleContextEL2 = CONTEXTIDR_EL2<31:0>;
        SPESampleContextEL2Valid = TRUE;
```

Library pseudocode for aarch64/debug/statisticalprofiling/SPESampleAddOpOther

```
// SPESampleAddOpOther()
// =====================
// Add other operation to sample storage.

SPESampleAddOpOther(boolean conditional, boolean taken)
    SPESampleEvents<6> = if conditional && !taken then '1' else '0';
    SPESampleAddOpOther(conditional);

SPESampleAddOpOther(boolean conditional)
    SPESampleClass = '00';
    SPESampleSubclass<0> = if conditional then '1' else '0';
    SPESampleOpType = OpType_Other;
```

Library pseudocode for aarch64/debug/statisticalprofiling/SPESampleAddOpSVELoadStore

```
// SPESampleAddOpSVELoadStore()
// ============================
// Sets the subclass of the operation type packet to Load/Store for SVE operations.

SPESampleAddOpSVELoadStore(boolean is_gather_scatter, bits(3) evl, boolean predicated,
                           boolean is_load)
    bit sg = if is_gather_scatter then '1' else '0';
    bit pred = if predicated then '1' else '0';
    bit ldst = if is_load then '0' else '1';
    SPESampleClass = '01';
    SPESampleSubclass<7:0> = sg:evl:'1':pred:'0':ldst;
    SPESampleSubclassValid = TRUE;
    SPESampleOpType = if is_load then OpType_Load else OpType_Store;
```

Library pseudocode for aarch64/debug/statisticalprofiling/SPESampleAddOpSVEOther

```
// SPESampleAddOpSVEOther()
// ========================
// Sets the subclass of the operation type packet to Other for SVE operations.

SPESampleAddOpSVEOther(bits(3) evl, boolean predicated, boolean floating_point)
    bit pred = if predicated then '1' else '0';
    bit fp = if floating_point then '1' else '0';
    SPESampleClass = '00';
    SPESampleSubclass<7:0> = '0':evl:'1':pred:fp:'0';
    SPESampleSubclassValid = TRUE;
    SPESampleOpType = OpType_Other;
```

Library pseudocode for aarch64/debug/statisticalprofiling/SPESampleAddTimeStamp

```
// SPESampleAddTimeStamp()
// =======================
// Save the appropriate type of timestamp to sample storage.

SPESampleAddTimeStamp()
    TimeStamp timestamp = CollectTimeStamp();
    case timestamp of
        when TimeStamp_None
            SPESampleTimestampValid = FALSE;
        otherwise
            SPESampleTimestampValid = TRUE;
            SPESampleTimestamp = GetTimestamp(timestamp);
```

Library pseudocode for aarch64/debug/statisticalprofiling/SPESampleExtendedLoadStore

```
// SPESampleExtendedLoadStore()
// ============================
// Sets the subclass of the operation type packet for
// extended load/store operations.

SPESampleExtendedLoadStore(bit ar, bit excl, bit at, boolean is_load)
    SPESampleClass = '01';
    bit ldst = if is_load then '0' else '1';
    SPESampleSubclass = '000':ar:excl:at:'1':ldst;

    SPESampleSubclassValid = TRUE;

    if is_load then
        if at == '1' then
            SPESampleOpType = OpType_LoadAtomic;
        else
            SPESampleOpType = OpType_Load;
    else
        SPESampleOpType = OpType_Store;
```

Library pseudocode for aarch64/debug/statisticalprofiling/SPESampleGeneralPurposeLoadStore

```
// SPESampleGeneralPurposeLoadStore()
// ==================================
// Sets the subclass of the operation type packet for general
// purpose load/store operations.

SPESampleGeneralPurposeLoadStore()
    SPESampleClass = '01';

    SPESampleSubclass<7:1> = Zeros(7);
    SPESampleSubclassValid = TRUE;
```

Library pseudocode for aarch64/debug/statisticalprofiling/SPESampleLoadStore

```
// SPESampleLoadStore()
// ====================
// Called if a sample is in flight when writing or reading memory,
// indicating that the operation being sampled is in the Load, Store or atomic category.

SPESampleLoadStore(boolean is_load, AccessDescriptor accdesc, AddressDescriptor addrdesc)
    // Check if this access type should be sampled.
    if accdesc.acctype IN {AccessType_SPE,
                           AccessType_IFETCH,
                           AccessType_DC,
                           AccessType_TTW,
                           AccessType_AT} then
        return;

    // MOPS instructions indicate which operation should be sampled before the
    // operation is executed. Has the instruction indicated that the load should be sampled?
    boolean sample_loads;
    sample_loads = SPESampleSubclass<0> == '0' && SPESampleSubclassValid;

    // Has the instruction indicated that the store should be sampled?
    boolean sample_stores;
    sample_stores = SPESampleSubclass<0> == '1' && SPESampleSubclassValid;

    // No valid data has been collected, or this is operation has specifically been selected for
    // sampling.
    if (!SPESampleSubclassValid || (sample_loads && is_load) ||
            (sample_stores && !is_load)) then
        // Data access virtual address
        SPESetDataVirtualAddress(addrdesc.vaddress);

        // Data access physical address
        if CollectPhysicalAddress() then
            SPESetDataPhysicalAddress(addrdesc, accdesc);

    if !SPESampleSubclassValid then
        // Set as unspecified load/store by default, instructions will overwrite this if it does not
        // apply to them.
        SPESampleClass = '01';
        SPESampleSubclassValid = TRUE;
        SPESampleSubclass<7:1> = '0001000';
        SPESampleSubclass<0> = if is_load then '0' else '1';
        SPESampleOpType = if is_load then OpType_Load else OpType_Store;

        if accdesc.acctype == AccessType_NV2 then
            // NV2 register load/store
            SPESampleSubclass<7:1> = '0011000';
            SPESampleInstIsNV2 = TRUE;
```

Library pseudocode for aarch64/debug/statisticalprofiling/SPESampleMemCopy

```
// SPESampleMemCopy()
// ==================
// Sets the subclass of the operation type packet for Memory Copy load/store
// operations.

SPESampleMemCopy()
    // MemCopy does a read and a write. If one is filtered out, the other should be recorded.
    // If neither or both are filtered out, pick one in a (pseudo)random way.

    // Are loads allowed by filter?
    boolean loads_pass_filter = PMSFCR_EL1.FT == '1' && PMSFCR_EL1.LD == '1';
    // Are stores allowed by filter?
    boolean stores_pass_filter = PMSFCR_EL1.FT == '1' && PMSFCR_EL1.ST == '1';

    boolean record_load;
    if loads_pass_filter && !stores_pass_filter then
        // Only loads pass filter
        record_load = TRUE;
    elsif !loads_pass_filter && stores_pass_filter then
        // Only stores pass filter
        record_load = FALSE;
    else
        // Pick randomly between
        record_load = SPEGetRandomBoolean();

    SPESampleClass = '01';
    bit ldst = if record_load then '0' else '1';
    SPESampleSubclass<7:0> = '0010000':ldst;
    SPESampleSubclassValid = TRUE;
    SPESampleOpType = if record_load then OpType_Load else OpType_Store;
```

Library pseudocode for aarch64/debug/statisticalprofiling/SPESampleMemSet

```
// SPESampleMemSet()
// =================
// Sets the subclass of the operation type packet for Memory Set load/store
// operation.

SPESampleMemSet()
    SPESampleClass = '01';
    SPESampleSubclass<7:0> = '00100101';
    SPESampleSubclassValid = TRUE;
    SPESampleOpType = OpType_Store;
```

Library pseudocode for aarch64/debug/statisticalprofiling/SPESampleSIMDFPLoadStore

```
// SPESampleSIMDFPLoadStore()
// ==========================
// Sets the subclass of the operation type packet for SIMD & FP
// load store operations.

SPESampleSIMDFPLoadStore()
    SPESampleClass = '01';

    SPESampleSubclass<7:1> = '0000010';
    SPESampleSubclassValid = TRUE;
```

Library pseudocode for aarch64/debug/statisticalprofiling/SPESetDataPhysicalAddress

```
// SPESetDataPhysicalAddress()
// ===========================
// Called from SampleLoadStore() to save data physical packet.

SPESetDataPhysicalAddress(AddressDescriptor addrdesc, AccessDescriptor accdesc)
    bit ns;
    bit nse;
    case addrdesc.paddress.paspace of
        when PAS_Secure
            ns = '0';
            nse = '0';
        when PAS_NonSecure
            ns = '1';
            nse = '0';
        when PAS_Realm
            ns  = '1';
            nse = '1';
        otherwise Unreachable();

    if HaveMTE2Ext() then
        bits(4) pat;
        if accdesc.tagchecked then
            SPESampleAddress[SPEAddrPosDataPhysical]<62> = '1'; // CH
            pat = AArch64.PhysicalTag(addrdesc.vaddress);
        else
            // CH is reset to 0 on each new packet
            // If the access is Unchecked, this is an IMPLEMENTATION_DEFINED choice
            // between 0b0000 and the Physical Address Tag
            boolean zero_unchecked;
            zero_unchecked = boolean IMPLEMENTATION_DEFINED "SPE PAT for tag unchecked access zero";
            if !zero_unchecked then
                pat = AArch64.PhysicalTag(addrdesc.vaddress);
            else
                pat = Zeros(4);
        SPESampleAddress[SPEAddrPosDataPhysical]<59:56> = pat;

    bits(56) paddr = addrdesc.paddress.address;
    SPESampleAddress[SPEAddrPosDataPhysical]<56-1:0> = paddr;
    SPESampleAddress[SPEAddrPosDataPhysical]<63> = ns;
    SPESampleAddress[SPEAddrPosDataPhysical]<60> = nse;
    SPESampleAddressValid[SPEAddrPosDataPhysical] = TRUE;
```

Library pseudocode for aarch64/debug/statisticalprofiling/SPESetDataVirtualAddress

```
// SPESetDataVirtualAddress()
// ==========================
// Called from SampleLoadStore() to save data virtual packet.
// Also used by exclusive load/stores to save virtual addresses if exclusive monitor is lost
// before a read/write is completed.

SPESetDataVirtualAddress(bits(64) vaddress)
    bit tbi;
    tbi = EffectiveTBI(vaddress, FALSE, PSTATE.EL);
    boolean non_tbi_is_zeros;
    non_tbi_is_zeros = boolean IMPLEMENTATION_DEFINED "SPE non-tbi tag is zero";
    if tbi == '1' || !non_tbi_is_zeros then
        SPESampleAddress[SPEAddrPosDataVirtual]<63:0> = vaddress<63:0>;
    else
        SPESampleAddress[SPEAddrPosDataVirtual]<63:56> = Zeros(8);
        SPESampleAddress[SPEAddrPosDataVirtual]<55:0> = vaddress<55:0>;
    SPESampleAddressValid[SPEAddrPosDataVirtual] = TRUE;
```

Library pseudocode for aarch64/debug/statisticalprofiling/SPEStartCounter

```
// SPEStartCounter()
// =================
// Enables incrementing of the counter at the passed index when SPECycle is called.

SPEStartCounter(integer counter_index)
    assert counter_index < SPEMaxCounters;
    SPESampleCounterPending[counter_index] = TRUE;
```

Library pseudocode for aarch64/debug/statisticalprofiling/SPEStopCounter

```
// SPEStopCounter()
// ================
// Disables incrementing of the counter at the passed index when SPECycle is called.

SPEStopCounter(integer counter_index)
    SPESampleCounterValid[counter_index] = TRUE;
    SPESampleCounterPending[counter_index] = FALSE;
```

Library pseudocode for aarch64/debug/statisticalprofiling/SPEToCollectSample

```
// SPEToCollectSample()
// ====================
// Returns TRUE if the instruction which is about to be executed should be
// sampled. Returns FALSE otherwise.

boolean SPEToCollectSample()
    if IsZero(PMSICR_EL1.COUNT) then
        SPEResetSampleCounter();
    else
        PMSICR_EL1.COUNT = PMSICR_EL1.COUNT - 1;
        if IsZero(PMSICR_EL1.COUNT) then
            if PMSIRR_EL1.RND == '1' && PMSIDR_EL1.ERnd == '1' then
                PMSICR_EL1.ECOUNT = SPEGetRandomInterval();
            else
                return TRUE;
    if UInt(PMSICR_EL1.ECOUNT) != 0 then
        PMSICR_EL1.ECOUNT = PMSICR_EL1.ECOUNT - 1;
        if IsZero(PMSICR_EL1.ECOUNT) then
            return TRUE;
    return FALSE;
```

Library pseudocode for aarch64/debug/statisticalprofiling/SPEWriteToBuffer

```
// SPEWriteToBuffer()
// ==================
// Write the active record to the Profiling Buffer.

SPEWriteToBuffer()
    assert ProfilingBufferEnabled();

    // Check alignment
    boolean aligned = IsZero(PMBPTR_EL1.PTR<UInt(PMBIDR_EL1.Align)-1:0>);
    boolean ttw_fault_as_external_abort;
    ttw_fault_as_external_abort = boolean IMPLEMENTATION_DEFINED "SPE TTW fault External abort";

    FaultRecord fault;
    PhysMemRetStatus memstatus;
    AddressDescriptor addrdesc;
    AccessDescriptor accdesc;

    SecurityState owning_ss;
    bits(2) owning_el;
    (owning_ss, owning_el) = ProfilingBufferOwner();
    accdesc = CreateAccDescSPE(owning_ss, owning_el);

    bits(64) start_vaddr = PMBPTR_EL1<63:0>;
    for i = 0 to SPERecordSize - 1
        // If a previous write did not cause an issue
        if PMBSR_EL1.S == '0' then
            (memstatus, addrdesc) = DebugMemWrite(PMBPTR_EL1<63:0>, accdesc, aligned,
                                                  SPERecordData[i]);

            fault = addrdesc.fault;

            boolean ttw_fault;
            ttw_fault = fault.statuscode IN {Fault_SyncExternalOnWalk, Fault_SyncParityOnWalk};

            if IsFault(fault.statuscode) && !(ttw_fault && ttw_fault_as_external_abort) then
                DebugWriteFault(PMBPTR_EL1<63:0>, fault);
            elsif IsFault(memstatus) || (ttw_fault && ttw_fault_as_external_abort) then
                DebugWriteExternalAbort(memstatus, addrdesc, start_vaddr);

            // Move pointer if no Buffer Management Event has been caused.
            if IsZero(PMBSR_EL1.S) then
                PMBPTR_EL1 = PMBPTR_EL1 + 1;

    return;
```

Library pseudocode for aarch64/debug/statisticalprofiling/StatisticalProfilingEnabled

```
// StatisticalProfilingEnabled()
// =============================
// Return TRUE if Statistical Profiling is Enabled in the current EL, FALSE otherwise.

boolean StatisticalProfilingEnabled()
    return StatisticalProfilingEnabled(PSTATE.EL);

// StatisticalProfilingEnabled()
// =============================
// Return TRUE if Statistical Profiling is Enabled in the specified EL, FALSE otherwise.

boolean StatisticalProfilingEnabled(bits(2) el)
    if !HaveStatisticalProfiling() || UsingAArch32() || !ProfilingBufferEnabled() then
        return FALSE;

    tge_set = EL2Enabled() && HCR_EL2.TGE == '1';
    (owning_ss, owning_el) = ProfilingBufferOwner();
    if (UInt(owning_el) <  UInt(el) || (tge_set && owning_el == EL1) ||
            owning_ss != SecurityStateAtEL(el))  then
        return FALSE;
    bit spe_bit;
    case el of
        when EL3  Unreachable();
        when EL2  spe_bit = PMSCR_EL2.E2SPE;
        when EL1  spe_bit = PMSCR_EL1.E1SPE;
        when EL0  spe_bit = (if tge_set then PMSCR_EL2.E0HSPE else PMSCR_EL1.E0SPE);

    return spe_bit == '1';
```

Library pseudocode for aarch64/debug/statisticalprofiling/TimeStamp

```
// TimeStamp
// =========

enumeration TimeStamp {
    TimeStamp_None,              // No timestamp
    TimeStamp_CoreSight,         // CoreSight time (IMPLEMENTATION DEFINED)
    TimeStamp_Physical,          // Physical counter value with no offset
    TimeStamp_OffsetPhysical,    // Physical counter value minus CNTPOFF_EL2
    TimeStamp_Virtual  };       // Physical counter value minus CNTVOFF_EL2
```

Library pseudocode for aarch64/debug/takeexceptiondbg/AArch64.TakeExceptionInDebugState

```
// AArch64.TakeExceptionInDebugState()
// ===================================
// Take an exception in Debug state to an Exception level using AArch64.

AArch64.TakeExceptionInDebugState(bits(2) target_el, ExceptionRecord exception_in)
    assert HaveEL(target_el) && !ELUsingAArch32(target_el) && UInt(target_el) >= UInt(PSTATE.EL);
    assert target_el != EL3 || EDSCR.SDD == '0';
    ExceptionRecord except = exception_in;
    boolean sync_errors;
    if HaveIESB() then
        sync_errors = SCTLR[target_el].IESB == '1';
        if HaveDoubleFaultExt() then
            sync_errors = sync_errors || (SCR_EL3.<EA,NMEA> == '11' && target_el == EL3);
        // SCTLR[].IESB and/or SCR_EL3.NMEA (if applicable) might be ignored in Debug state.
        if !ConstrainUnpredictableBool(Unpredictable_IESBinDebug) then
            sync_errors = FALSE;
    else
        sync_errors = FALSE;

    if HaveTME() && TSTATE.depth > 0 then
        TMFailure cause;
        case except.exceptype of
            when Exception_SoftwareBreakpoint cause = TMFailure_DBG;
            when Exception_Breakpoint         cause = TMFailure_DBG;
            when Exception_Watchpoint         cause = TMFailure_DBG;
            when Exception_SoftwareStep       cause = TMFailure_DBG;
            otherwise                         cause = TMFailure_ERR;
        FailTransaction(cause, FALSE);

    SynchronizeContext();

    // If coming from AArch32 state, the top parts of the X[] registers might be set to zero
    from_32 = UsingAArch32();
    if from_32 then AArch64.MaybeZeroRegisterUppers();
    if from_32 && HaveSME() && PSTATE.SM == '1' then
        ResetSVEState();
    else
        MaybeZeroSVEUppers(target_el);

    AArch64.ReportException(except, target_el);

    if HaveGCS() then
        PSTATE.EXLOCK = '0';  // Effective value of GCSCR_ELx.EXLOCKEN is 0 in Debug state

    PSTATE.EL = target_el;
    PSTATE.nRW = '0';
    PSTATE.SP = '1';

    SPSR[] = bits(64) UNKNOWN;
    ELR[] = bits(64) UNKNOWN;

    // PSTATE.{SS,D,A,I,F} are not observable and ignored in Debug state, so behave as if UNKNOWN.
    PSTATE.<SS,D,A,I,F> = bits(5) UNKNOWN;
    PSTATE.IL = '0';
    if from_32 then                             // Coming from AArch32
        PSTATE.IT = '00000000';
        PSTATE.T = '0';                         // PSTATE.J is RES0
    if (HavePANExt() && (PSTATE.EL == EL1 || (PSTATE.EL == EL2 && ELIsInHost(EL0))) &&
        SCTLR[].SPAN == '0') then
        PSTATE.PAN = '1';
    if HaveUAOExt() then PSTATE.UAO = '0';
    if HaveBTIExt() then PSTATE.BTYPE = '00';
    if HaveSSBSExt() then PSTATE.SSBS = bit UNKNOWN;
    if HaveMTEExt() then PSTATE.TCO = '1';

    DLR_EL0 = bits(64) UNKNOWN;
    DSPSR_EL0 = bits(64) UNKNOWN;

    EDSCR.ERR = '1';
    UpdateEDSCRFields();                        // Update EDSCR processor state flags.
```

```
    if sync_errors then
        SynchronizeErrors();

    EndOfInstruction();
```

Library pseudocode for aarch64/debug/watchpoint/AArch64.WatchpointByteMatch

```
// AArch64.WatchpointByteMatch()
// =============================

boolean AArch64.WatchpointByteMatch(integer n, bits(64) vaddress)

    integer top = DebugAddrTop();
    bottom = if DBGWVR_EL1[n]<2> == '1' then 2 else 3;            // Word or doubleword
    byte_select_match = (DBGWCR_EL1[n].BAS<UInt(vaddress<bottom-1:0>)> != '0');
    mask = UInt(DBGWCR_EL1[n].MASK);

    // If DBGWCR_EL1[n].MASK is a nonzero value and DBGWCR_EL1[n].BAS is not set to '11111111', or
    // DBGWCR_EL1[n].BAS specifies a non-contiguous set of bytes behavior is CONSTRAINED
    // UNPREDICTABLE.
    if mask > 0 && !IsOnes(DBGWCR_EL1[n].BAS) then
        byte_select_match = ConstrainUnpredictableBool(Unpredictable_WPMASKANDBAS);
    else
        LSB = (DBGWCR_EL1[n].BAS AND NOT(DBGWCR_EL1[n].BAS - 1));  MSB = (DBGWCR_EL1[n].BAS + LSB);
        if !IsZero(MSB AND (MSB - 1)) then                     // Not contiguous
            byte_select_match = ConstrainUnpredictableBool(Unpredictable_WPBASCONTIGUOUS);
            bottom = 3;                                        // For the whole doubleword

    // If the address mask is set to a reserved value, the behavior is CONSTRAINED UNPREDICTABLE.
    if mask > 0 && mask <= 2 then
        Constraint c;
        (c, mask) = ConstrainUnpredictableInteger(3, 31, Unpredictable_RESWPMASK);
        assert c IN {Constraint_DISABLED, Constraint_NONE, Constraint_UNKNOWN};
        case c of
            when Constraint_DISABLED  return FALSE;            // Disabled
            when Constraint_NONE      mask = 0;                // No masking
            // Otherwise the value returned by ConstrainUnpredictableInteger is a not-reserved value

    // When FEAT_LVA3 is not implemented, if the DBGWVR_EL1[n].RESS field bits are not a
    // sign extension of the MSB of DBGWVR_EL1[n].VA, it is UNPREDICTABLE whether they
    // appear to be included in the match.
    if (top < 55 && !IsOnes(DBGWVR_EL1[n]<63:top>) && !IsZero(DBGWVR_EL1[n]<63:top>) &&
          ConstrainUnpredictableBool(Unpredictable_DBGxVR_RESS)) then
        top = 63;

    boolean WVR_match;
    if mask > bottom then
        WVR_match = (vaddress<top:mask> == DBGWVR_EL1[n]<top:mask>);
        // If masked bits of DBGWVR_EL1[n] are not zero, the behavior is CONSTRAINED UNPREDICTABLE.
        if WVR_match && !IsZero(DBGWVR_EL1[n]<mask-1:bottom>) then
            WVR_match = ConstrainUnpredictableBool(Unpredictable_WPMASKEDBITS);
    else
        WVR_match = vaddress<top:bottom> == DBGWVR_EL1[n]<top:bottom>;

    return WVR_match && byte_select_match;
```

Library pseudocode for aarch64/debug/watchpoint/AArch64.WatchpointMatch

```
// AArch64.WatchpointMatch()
// =========================
// Watchpoint matching in an AArch64 translation regime.

boolean AArch64.WatchpointMatch(integer n, bits(64) vaddress, integer size,
                                AccessDescriptor accdesc)
    assert !ELUsingAArch32(S1TranslationRegime());
    assert n < NumWatchpointsImplemented();

    enabled = IsWatchpointEnabled(n);
    linked = DBGWCR_EL1[n].WT == '1';
    isbreakpnt = FALSE;
    lbnx = if Havev8p9Debug() then DBGWCR_EL1[n].LBNX else '00';
    linked_n = UInt(lbnx : DBGWCR_EL1[n].LBN);
    ssce = if HaveRME() then DBGWCR_EL1[n].SSCE else '0';
    state_match = AArch64.StateMatch(DBGWCR_EL1[n].SSC, ssce, DBGWCR_EL1[n].HMC, DBGWCR_EL1[n].PAC,
                                     linked, linked_n, isbreakpnt, PC[], accdesc);

    boolean ls_match;
    case DBGWCR_EL1[n].LSC<1:0> of
        when '00' ls_match = FALSE;
        when '01' ls_match = accdesc.read;
        when '10' ls_match = accdesc.write || accdesc.acctype == AccessType_DC;
        when '11' ls_match = TRUE;

    value_match = FALSE;
    for byte = 0 to size - 1
        value_match = value_match || AArch64.WatchpointByteMatch(n, vaddress + byte);

    return value_match && state_match && ls_match && enabled;
```

Library pseudocode for aarch64/debug/watchpoint/IsWatchpointEnabled

```
// IsWatchpointEnabled()
// =====================
// Returns TRUE if the effective value of DBGWCR_EL1[n].E is '1', and FALSE otherwise.

boolean IsWatchpointEnabled(integer n)
    if (n > 15 &&
          ((!HaltOnBreakpointOrWatchpoint() && !SelfHostedExtendedBPWPEnabled()) ||
           (HaltOnBreakpointOrWatchpoint() && EDSCR2.EHBWE == '0'))) then
        return FALSE;
    return DBGWCR_EL1[n].E == '1';
```

Library pseudocode for aarch64/exceptions/aborts/AArch64.Abort

```
// AArch64.Abort()
// ===============
// Abort and Debug exception handling in an AArch64 translation regime.

AArch64.Abort(bits(64) vaddress, FaultRecord fault)

    if IsDebugException(fault) then
        if fault.access.acctype == AccessType_IFETCH then
            if UsingAArch32() && fault.debugmoe == DebugException_VectorCatch then
                AArch64.VectorCatchException(fault);
            else
                AArch64.BreakpointException(fault);
        else
            AArch64.WatchpointException(vaddress, fault);
    elsif fault.gpcf.gpf != GPCF_None && ReportAsGPCException(fault) then
        TakeGPCException(vaddress, fault);
    elsif fault.access.acctype == AccessType_IFETCH then
        AArch64.InstructionAbort(vaddress, fault);
    else
        AArch64.DataAbort(vaddress, fault);
```

Library pseudocode for aarch64/exceptions/aborts/AArch64.AbortSyndrome

```
// AArch64.AbortSyndrome()
// =======================
// Creates an exception syndrome record for Abort and Watchpoint exceptions
//
// from an AArch64 translation regime.

ExceptionRecord AArch64.AbortSyndrome(Exception exceptype, FaultRecord fault,
                                      bits(64) vaddress, bits(2) target_el)
    except = ExceptionSyndrome(exceptype);

    d_side = exceptype IN {Exception_DataAbort, Exception_NV2DataAbort, Exception_Watchpoint,
                           Exception_NV2Watchpoint};
    if (!HavePFAR() ||
            !IsExternalSyncAbort(fault) ||
            (EL2Enabled() && HCR_EL2.VM == '1' && target_el == EL1)) then
        except.pavalid = FALSE;
    else
        except.pavalid = boolean IMPLEMENTATION_DEFINED "PFAR_ELx is valid";

    (except.syndrome, except.syndrome2) = AArch64.FaultSyndrome(d_side, fault,
                                                                except.pavalid);
    if fault.statuscode == Fault_TagCheck then
        if HaveMTE4Ext() then
            except.vaddress = ZeroExtend(vaddress, 64);
        else
            except.vaddress = bits(4) UNKNOWN : vaddress<59:0>;
    else
        except.vaddress = ZeroExtend(vaddress, 64);

    if IPAValid(fault) then
        except.ipavalid = TRUE;
        except.NS = if fault.ipaddress.paspace == PAS_NonSecure then '1' else '0';
        except.ipaddress = fault.ipaddress.address;
    else
        except.ipavalid = FALSE;

    return except;
```

Library pseudocode for aarch64/exceptions/aborts/AArch64.CheckPCAlignment

```
// AArch64.CheckPCAlignment()
// ==========================

AArch64.CheckPCAlignment()
    bits(64) pc = ThisInstrAddr(64);

    if pc<1:0> != '00' then
        AArch64.PCAlignmentFault();
```

Library pseudocode for aarch64/exceptions/aborts/AArch64.DataAbort

```
// AArch64.DataAbort()
// ===================

AArch64.DataAbort(bits(64) vaddress, FaultRecord fault)
    bits(2) target_el;
    if IsExternalAbort(fault) then
        target_el = AArch64.SyncExternalAbortTarget(fault);
    else
        route_to_el2 = (EL2Enabled() && PSTATE.EL IN {EL0, EL1} &&
                        (HCR_EL2.TGE == '1' ||
                         (HaveRME() && fault.gpcf.gpf == GPCF_Fail && HCR_EL2.GPF == '1') ||
                         (HaveNV2Ext() && fault.access.acctype == AccessType_NV2) ||
                         IsSecondStage(fault)));

        if PSTATE.EL == EL3 then
            target_el = EL3;
        elsif PSTATE.EL == EL2 || route_to_el2 then
            target_el = EL2;
        else
            target_el = EL1;

    bits(64) preferred_exception_return = ThisInstrAddr(64);
    integer vect_offset;

    if IsExternalAbort(fault) && AArch64.RouteToSErrorOffset(target_el) then
        vect_offset = 0x180;
    else
        vect_offset = 0x0;

    ExceptionRecord except;
    if HaveNV2Ext() && fault.access.acctype == AccessType_NV2 then
        except = AArch64.AbortSyndrome(Exception_NV2DataAbort, fault, vaddress, target_el);
    else
        except = AArch64.AbortSyndrome(Exception_DataAbort, fault, vaddress, target_el);
    AArch64.TakeException(target_el, except, preferred_exception_return, vect_offset);
```

Library pseudocode for aarch64/exceptions/aborts/AArch64.EffectiveTCF

```
// AArch64.EffectiveTCF()
// ======================
// Returns the TCF field applied to tag check faults in the given Exception level.

bits(2) AArch64.EffectiveTCF(bits(2) el)
    bits(2) tcf;

    Regime regime = TranslationRegime(el);

    case regime of
        when Regime_EL3  tcf = SCTLR_EL3.TCF;
        when Regime_EL2  tcf = SCTLR_EL2.TCF;
        when Regime_EL20 tcf = if el == EL0 then SCTLR_EL2.TCF0 else SCTLR_EL2.TCF;
        when Regime_EL10 tcf = if el == EL0 then SCTLR_EL1.TCF0 else SCTLR_EL1.TCF;
        otherwise        Unreachable();

    if tcf == '11' then        //reserved value
        if !HaveMTEAsymFaultExt() then
            (-,tcf) = ConstrainUnpredictableBits(Unpredictable_RESTCF, 2);

    return tcf;
```

Library pseudocode for aarch64/exceptions/aborts/AArch64.InstructionAbort

```
// AArch64.InstructionAbort()
// ==========================

AArch64.InstructionAbort(bits(64) vaddress, FaultRecord fault)
    // External aborts on instruction fetch must be taken synchronously
    if HaveDoubleFaultExt() then assert fault.statuscode != Fault_AsyncExternal;

    bits(2) target_el;
    if IsExternalAbort(fault) then
        target_el = AArch64.SyncExternalAbortTarget(fault);
    else
        route_to_el2 = (EL2Enabled() && PSTATE.EL IN {EL0, EL1} &&
                        (HCR_EL2.TGE == '1' ||
                         (HaveRME() && fault.gpcf.gpf == GPCF_Fail && HCR_EL2.GPF == '1') ||
                         IsSecondStage(fault)));

        if PSTATE.EL == EL3 then
            target_el = EL3;
        elsif PSTATE.EL == EL2 || route_to_el2 then
            target_el = EL2;
        else
            target_el = EL1;

    bits(64) preferred_exception_return = ThisInstrAddr(64);
    integer vect_offset;

    if IsExternalAbort(fault) && AArch64.RouteToSErrorOffset(target_el) then
        vect_offset = 0x180;
    else
        vect_offset = 0x0;

    ExceptionRecord except = AArch64.AbortSyndrome(Exception_InstructionAbort, fault,
                                                   vaddress, target_el);
    AArch64.TakeException(target_el, except, preferred_exception_return, vect_offset);
```

Library pseudocode for aarch64/exceptions/aborts/AArch64.PCAlignmentFault

```
// AArch64.PCAlignmentFault()
// ==========================
// Called on unaligned program counter in AArch64 state.

AArch64.PCAlignmentFault()

    bits(64) preferred_exception_return = ThisInstrAddr(64);
    vect_offset = 0x0;

    except = ExceptionSyndrome(Exception_PCAlignment);
    except.vaddress = ThisInstrAddr(64);
    bits(2) target_el = EL1;
    if UInt(PSTATE.EL) > UInt(EL1) then
        target_el = PSTATE.EL;
    elsif EL2Enabled() && HCR_EL2.TGE == '1' then
        target_el = EL2;
    AArch64.TakeException(target_el, except, preferred_exception_return, vect_offset);
```

Library pseudocode for aarch64/exceptions/aborts/AArch64.PhysicalSErrorTarget

```
// AArch64.PhysicalSErrorTarget()
// ==============================
// Returns a tuple of whether SError exception can be taken and, if so, the target Exception level.

(boolean, bits(2)) AArch64.PhysicalSErrorTarget()
    boolean route_to_el3;
    boolean route_to_el2;

    // The exception is explicitly routed to EL3.
    if PSTATE.EL != EL3 then
        route_to_el3 = (HaveEL(EL3) && EffectiveEA() == '1');
    else
        route_to_el3 = FALSE;

    // The exception is explicitly routed to EL2.
    if !route_to_el3 && EL2Enabled() && PSTATE.EL == EL1 then
        route_to_el2 = (HCR_EL2.AMO == '1');
    elsif !route_to_el3 && EL2Enabled() && PSTATE.EL == EL0 then
        route_to_el2 = (!IsInHost() && HCR_EL2.<TGE,AMO> != '00');
    else
        route_to_el2 = FALSE;

    // The exception is "masked".
    boolean masked;
    case PSTATE.EL of
        when EL3
            masked = (EffectiveEA()  == '0' || PSTATE.A == '1');
        when EL2
            masked = (!route_to_el3 &&
                      (HCR_EL2.<TGE,AMO> == '00' || PSTATE.A == '1'));
        when EL1, EL0
            masked = (!route_to_el3 && !route_to_el2 && PSTATE.A == '1');

    // When FEAT_DoubleFault or FEAT_DoubleFault2 is implemented, the mask might be overridden.
    if HaveDoubleFault2Ext() then
        bit nmea_bit;
        case PSTATE.EL of
            when EL3
                nmea_bit = SCR_EL3.NMEA;
            when EL2
                nmea_bit = if IsSCTLR2EL2Enabled() then SCTLR2_EL2.NMEA else '0';
            when EL1
                nmea_bit = if IsSCTLR2EL1Enabled() then SCTLR2_EL1.NMEA else '0';
            when EL0
                if IsInHost() then
                    nmea_bit = if IsSCTLR2EL2Enabled() then SCTLR2_EL2.NMEA else '0';
                else
                    nmea_bit = if IsSCTLR2EL1Enabled() then SCTLR2_EL1.NMEA else '0';
        masked = masked && (nmea_bit == '0');

    elsif HaveDoubleFaultExt() && PSTATE.EL == EL3 then
        bit nmea_bit = SCR_EL3.NMEA AND EffectiveEA();
        masked = masked && (nmea_bit == '0');

    boolean route_masked_to_el3;
    boolean route_masked_to_el2;

    if HaveDoubleFault2Ext() then
        // The masked exception is routed to EL2.
        route_masked_to_el2 = (EL2Enabled() && !route_to_el3 &&
                               IsHCRXEL2Enabled() && HCRX_EL2.TMEA == '1' &&
                               ((PSTATE.EL == EL1 && (PSTATE.A == '1' || masked)) ||
                               (PSTATE.EL == EL0 && masked && !IsInHost())));

        // The masked exception is routed to EL3.
        route_masked_to_el3 = (HaveEL(EL3) && SCR_EL3.TMEA == '1' &&
                               !(route_to_el2 || route_masked_to_el2) &&
                               ((PSTATE.EL IN {EL2, EL1} &&
                                   (PSTATE.A == '1' || masked)) ||
                                   (PSTATE.EL == EL0 && masked)));
```

```
    else
        route_masked_to_el2 = FALSE;
        route_masked_to_el3 = FALSE;

    // The exception is taken at EL3.
    take_in_el3 = PSTATE.EL == EL3 && !masked;

    // The exception is taken at EL2 or in the Host EL0.
    take_in_el2_0 = ((PSTATE.EL == EL2 || IsInHost()) &&
                     !(route_to_el3 || route_masked_to_el3) && !masked);

    // The exception is taken at EL1 or in the non-Host EL0.
    take_in_el1_0 = ((PSTATE.EL == EL1 || (PSTATE.EL == EL0 && !IsInHost())) &&
                     !(route_to_el2 || route_masked_to_el2) &&
                     !(route_to_el3 || route_masked_to_el3) && !masked);

    bits(2) target_el;
    if take_in_el3 || route_to_el3 || route_masked_to_el3 then
        masked = FALSE;  target_el = EL3;
    elsif take_in_el2_0 || route_to_el2 || route_masked_to_el2 then
        masked = FALSE;  target_el = EL2;
    elsif take_in_el1_0 then
        masked = FALSE;  target_el = EL1;
    else
        masked = TRUE;  target_el = bits(2) UNKNOWN;

    return (masked, target_el);
```

Library pseudocode for aarch64/exceptions/aborts/AArch64.RaiseTagCheckFault

```
// AArch64.RaiseTagCheckFault()
// ============================
// Raise a tag check fault exception.

AArch64.RaiseTagCheckFault(bits(64) va, FaultRecord fault)
    bits(64) preferred_exception_return = ThisInstrAddr(64);
    integer vect_offset = 0x0;
    bits(2) target_el = EL1;
    if UInt(PSTATE.EL) > UInt(EL1) then
        target_el = PSTATE.EL;
    elsif PSTATE.EL == EL0 && EL2Enabled() && HCR_EL2.TGE == '1' then
        target_el = EL2;

    except = AArch64.AbortSyndrome(Exception_DataAbort, fault, va, target_el);
    AArch64.TakeException(target_el, except, preferred_exception_return, vect_offset);
```

Library pseudocode for aarch64/exceptions/aborts/AArch64.ReportTagCheckFault

```
// AArch64.ReportTagCheckFault()
// =============================
// Records a tag check fault exception into the appropriate TFSR_ELx.

AArch64.ReportTagCheckFault(bits(2) el, bit ttbr)
    case el of
        when EL3 assert ttbr == '0'; TFSR_EL3.TF0   = '1';
        when EL2 if ttbr == '0' then TFSR_EL2.TF0   = '1'; else TFSR_EL2.TF1   = '1';
        when EL1 if ttbr == '0' then TFSR_EL1.TF0   = '1'; else TFSR_EL1.TF1   = '1';
        when EL0 if ttbr == '0' then TFSRE0_EL1.TF0 = '1'; else TFSRE0_EL1.TF1 = '1';
```

Library pseudocode for aarch64/exceptions/aborts/AArch64.RouteToSErrorOffset

```
// AArch64.RouteToSErrorOffset()
// =============================
// Returns TRUE if synchronous External abort exceptions are taken to the
// appropriate SError vector offset, and FALSE otherwise.

boolean AArch64.RouteToSErrorOffset(bits(2) target_el)
    if !HaveDoubleFaultExt() then return FALSE;

    bit ease_bit;
    case target_el of
        when EL3
            ease_bit = SCR_EL3.EASE;
        when EL2
            if HaveDoubleFault2Ext() && IsSCTLR2EL2Enabled() then
                ease_bit = SCTLR2_EL2.EASE;
            else
                ease_bit = '0';
        when EL1
            if HaveDoubleFault2Ext() && IsSCTLR2EL1Enabled() then
                ease_bit = SCTLR2_EL1.EASE;
            else
                ease_bit = '0';

    return (ease_bit == '1');
```

Library pseudocode for aarch64/exceptions/aborts/AArch64.SPAlignmentFault

```
// AArch64.SPAlignmentFault()
// ==========================
// Called on an unaligned stack pointer in AArch64 state.

AArch64.SPAlignmentFault()

    bits(64) preferred_exception_return = ThisInstrAddr(64);
    vect_offset = 0x0;

    except = ExceptionSyndrome(Exception_SPAlignment);

    bits(2) target_el = EL1;
    if UInt(PSTATE.EL) > UInt(EL1) then
        target_el = PSTATE.EL;
    elsif EL2Enabled() && HCR_EL2.TGE == '1' then
        target_el = EL2;
    AArch64.TakeException(target_el, except, preferred_exception_return, vect_offset);
```

Library pseudocode for aarch64/exceptions/aborts/AArch64.SyncExternalAbortTarget

```
// AArch64.SyncExternalAbortTarget()
// =================================
// Returns the target Exception level for a Synchronous External
// Data or Instruction Abort.

bits(2) AArch64.SyncExternalAbortTarget(FaultRecord fault)
    boolean route_to_el3;

    // The exception is explicitly routed to EL3
    if PSTATE.EL != EL3 then
        route_to_el3 = (HaveEL(EL3) && EffectiveEA() == '1');
    else
        route_to_el3 = FALSE;

    // The exception is explicitly routed to EL2
    bit tea_bit = (if HaveRASExt() then HCR_EL2.TEA else '0');

    boolean route_to_el2;
    if !route_to_el3 && EL2Enabled() && PSTATE.EL == EL1 then
        route_to_el2 = (tea_bit == '1' ||
                        fault.access.acctype == AccessType_NV2 ||
                        IsSecondStage(fault));

    elsif !route_to_el3 && EL2Enabled() && PSTATE.EL == EL0 then
        route_to_el2 = (!IsInHost() && (HCR_EL2.TGE == '1' || tea_bit == '1' ||
                        IsSecondStage(fault)));
    else
        route_to_el2 = FALSE;

    boolean route_masked_to_el3;
    boolean route_masked_to_el2;

    if HaveDoubleFault2Ext() then
        // The masked exception is routed to EL2
        route_masked_to_el2 = (EL2Enabled() && !route_to_el3 &&
                               (PSTATE.EL == EL1 && PSTATE.A == '1') &&
                               IsHCRXEL2Enabled() && HCRX_EL2.TMEA == '1');

        // The masked exception is routed to EL3
        route_masked_to_el3 = (HaveEL(EL3) &&
                               !(route_to_el2 || route_masked_to_el2) &&
                               (PSTATE.EL IN {EL2, EL1} && PSTATE.A == '1') &&
                               SCR_EL3.TMEA == '1');
    else
        route_masked_to_el2 = FALSE;
        route_masked_to_el3 = FALSE;

    // The exception is taken at EL3
    take_in_el3 = PSTATE.EL == EL3;

    // The exception is taken at EL2 or in the Host EL0
    take_in_el2_0 = ((PSTATE.EL == EL2 || IsInHost()) &&
                     !(route_to_el3 || route_masked_to_el3));

    // The exception is taken at EL1 or in the non-Host EL0
    take_in_el1_0 = ((PSTATE.EL == EL1 || (PSTATE.EL == EL0 && !IsInHost())) &&
                     !(route_to_el2 || route_masked_to_el2) &&
                     !(route_to_el3 || route_masked_to_el3));

    bits(2) target_el;
    if take_in_el3 || route_to_el3 || route_masked_to_el3 then
        target_el = EL3;
    elsif take_in_el2_0 || route_to_el2 || route_masked_to_el2 then
        target_el = EL2;
    elsif take_in_el1_0 then
        target_el = EL1;
    else
        assert(FALSE);

    return target_el;
```

Library pseudocode for aarch64/exceptions/aborts/AArch64.TagCheckFault

```
// AArch64.TagCheckFault()
// =======================
// Handle a tag check fault condition.

AArch64.TagCheckFault(bits(64) vaddress, AccessDescriptor accdesc)
    bits(2) tcf;

    tcf = AArch64.EffectiveTCF(accdesc.el);

    fault = NoFault(accdesc);
    fault.statuscode = Fault_TagCheck;

    case tcf of
        when '00'   // Tag Check Faults have no effect on the PE
            return;
        when '01'   // Tag Check Faults cause a synchronous exception
            AArch64.RaiseTagCheckFault(vaddress, fault);
        when '10'
            if HaveMTEAsyncExt() then
                // If asynchronous faults are implemented,
                // Tag Check Faults are asynchronously accumulated
                AArch64.ReportTagCheckFault(accdesc.el, vaddress<55>);
            else
                // Otherwise, Tag Check Faults have no effect on the PE.
                return;
        when '11'
            if HaveMTEAsymFaultExt() then
                // Tag Check Faults cause a synchronous exception on reads or on
                // a read/write access, and are asynchronously accumulated on writes
                if accdesc.read then
                    AArch64.RaiseTagCheckFault(vaddress, fault);
                else
                    AArch64.ReportTagCheckFault(accdesc.el, vaddress<55>);
            else
                // Otherwise, Tag Check Faults have no effect on the PE.
                return;
```

Library pseudocode for aarch64/exceptions/aborts/BranchTargetException

```
// BranchTargetException()
// =======================
// Raise branch target exception.

AArch64.BranchTargetException(bits(52) vaddress)
    bits(64) preferred_exception_return = ThisInstrAddr(64);
    vect_offset = 0x0;

    except = ExceptionSyndrome(Exception_BranchTarget);
    except.syndrome<1:0>   = PSTATE.BTYPE;
    except.syndrome<24:2>  = Zeros(23);         // RES0

    bits(2) target_el = EL1;
    if UInt(PSTATE.EL) > UInt(EL1) then
        target_el = PSTATE.EL;
    elsif PSTATE.EL == EL0 && EL2Enabled() && HCR_EL2.TGE == '1' then
        target_el = EL2;
    AArch64.TakeException(target_el, except, preferred_exception_return, vect_offset);
```

Library pseudocode for aarch64/exceptions/aborts/TakeGPCException

```
// TakeGPCException()
// ==================
// Report Granule Protection Exception faults

TakeGPCException(bits(64) vaddress, FaultRecord fault)
    assert HaveRME();
    assert HaveAtomicExt();
    assert HaveAccessFlagUpdateExt();
    assert HaveDirtyBitModifierExt();
    assert HaveDoubleFaultExt();

    ExceptionRecord except;

    except.exceptype = Exception_GPC;
    except.vaddress  = ZeroExtend(vaddress, 64);
    except.paddress  = fault.paddress;
    except.pavalid   = TRUE;

    if IPAValid(fault) then
        except.ipavalid  = TRUE;
        except.NS        = if fault.ipaddress.paspace == PAS_NonSecure then '1' else '0';
        except.ipaddress = fault.ipaddress.address;
    else
        except.ipavalid = FALSE;

    if fault.access.acctype == AccessType_GCS then
        except.syndrome2<8> = '1'; //GCS

    // Populate the fields grouped in ISS
    except.syndrome<24:22> = Zeros(3); // RES0
    except.syndrome<21>    = if fault.gpcfs2walk then '1' else '0';  // S2PTW
    if fault.access.acctype == AccessType_IFETCH then
        except.syndrome<20> = '1';     // InD
    else
        except.syndrome<20> = '0';     // InD
    except.syndrome<19:14> = EncodeGPCSC(fault.gpcf); // GPCSC
    if HaveNV2Ext() && fault.access.acctype == AccessType_NV2 then
        except.syndrome<13> = '1';     // VNCR
    else
        except.syndrome<13> = '0';     // VNCR
    except.syndrome<12:11> = '00';     // RES0
    except.syndrome<10:9>  = '00';     // RES0

    if fault.access.acctype IN {AccessType_DC, AccessType_IC, AccessType_AT} then
        except.syndrome<8> = '1'; // CM
    else
        except.syndrome<8> = '0';  // CM

    except.syndrome<7> = if fault.s2fs1walk then '1' else '0'; // S1PTW

    if fault.access.acctype IN {AccessType_DC, AccessType_IC, AccessType_AT} then
        except.syndrome<6> = '1';                            // WnR
    elsif fault.statuscode IN {Fault_HWUpdateAccessFlag, Fault_Exclusive} then
        except.syndrome<6> = bit UNKNOWN;                    // WnR
    elsif fault.access.atomicop && IsExternalAbort(fault) then
        except.syndrome<6> = bit UNKNOWN;                    // WnR
    else
        except.syndrome<6> = if fault.write then '1' else '0'; // WnR

    except.syndrome<5:0> = EncodeLDFSC(fault.statuscode, fault.level); // xFSC

    bits(64) preferred_exception_return = ThisInstrAddr(64);
    bits(2) target_el = EL3;

    integer vect_offset;
    if IsExternalAbort(fault) && AArch64.RouteToSErrorOffset(target_el) then
        vect_offset = 0x180;
    else
        vect_offset = 0x0;
```

```
    AArch64.TakeException(target_el, except, preferred_exception_return, vect_offset);
```

Library pseudocode for aarch64/exceptions/async/AArch64.TakePhysicalFIQException

```
// AArch64.TakePhysicalFIQException()
// ==================================

AArch64.TakePhysicalFIQException()

    route_to_el3 = HaveEL(EL3) && SCR_EL3.FIQ == '1';
    route_to_el2 = (PSTATE.EL IN {EL0, EL1} && EL2Enabled() &&
                    (HCR_EL2.TGE == '1' || HCR_EL2.FMO == '1'));
    bits(64) preferred_exception_return = ThisInstrAddr(64);
    vect_offset = 0x100;
    except = ExceptionSyndrome(Exception_FIQ);

    if route_to_el3 then
        AArch64.TakeException(EL3, except, preferred_exception_return, vect_offset);
    elsif PSTATE.EL == EL2 || route_to_el2 then
        assert PSTATE.EL != EL3;
        AArch64.TakeException(EL2, except, preferred_exception_return, vect_offset);
    else
        assert PSTATE.EL IN {EL0, EL1};
        AArch64.TakeException(EL1, except, preferred_exception_return, vect_offset);
```

Library pseudocode for aarch64/exceptions/async/AArch64.TakePhysicalIRQException

```
// AArch64.TakePhysicalIRQException()
// ==================================
// Take an enabled physical IRQ exception.

AArch64.TakePhysicalIRQException()

    route_to_el3 = HaveEL(EL3) && SCR_EL3.IRQ == '1';
    route_to_el2 = (PSTATE.EL IN {EL0, EL1} && EL2Enabled() &&
                    (HCR_EL2.TGE == '1' || HCR_EL2.IMO == '1'));
    bits(64) preferred_exception_return = ThisInstrAddr(64);
    vect_offset = 0x80;

    except = ExceptionSyndrome(Exception_IRQ);

    if route_to_el3 then
        AArch64.TakeException(EL3, except, preferred_exception_return, vect_offset);
    elsif PSTATE.EL == EL2 || route_to_el2 then
        assert PSTATE.EL != EL3;
        AArch64.TakeException(EL2, except, preferred_exception_return, vect_offset);
    else
        assert PSTATE.EL IN {EL0, EL1};
        AArch64.TakeException(EL1, except, preferred_exception_return, vect_offset);
```

Library pseudocode for aarch64/exceptions/async/AArch64.TakePhysicalSErrorException

```
// AArch64.TakePhysicalSErrorException()
// =====================================

AArch64.TakePhysicalSErrorException(boolean implicit_esb)

    route_to_el3 = HaveEL(EL3) && SCR_EL3.EA == '1';
    route_to_el2 = (PSTATE.EL IN {EL0, EL1} && EL2Enabled() &&
                    (HCR_EL2.TGE == '1' || (!IsInHost() && HCR_EL2.AMO == '1')));
    bits(64) preferred_exception_return = ThisInstrAddr(64);
    vect_offset = 0x180;

    bits(2) target_el;
    if PSTATE.EL == EL3 || route_to_el3 then
        target_el = EL3;
    elsif PSTATE.EL == EL2 || route_to_el2 then
        target_el = EL2;
    else
        target_el = EL1;

    except = ExceptionSyndrome(Exception_SError);
    bits(25) syndrome = AArch64.PhysicalSErrorSyndrome(implicit_esb);
    if IsSErrorEdgeTriggered() then
        ClearPendingPhysicalSError();
    except.syndrome = syndrome;
    AArch64.TakeException(target_el, except, preferred_exception_return, vect_offset);
```

Library pseudocode for aarch64/exceptions/async/AArch64.TakeVirtualFIQException

```
// AArch64.TakeVirtualFIQException()
// =================================

AArch64.TakeVirtualFIQException()
    assert PSTATE.EL IN {EL0, EL1} && EL2Enabled();
    assert HCR_EL2.TGE == '0' && HCR_EL2.FMO == '1';  // Virtual IRQ enabled if TGE==0 and FMO==1

    bits(64) preferred_exception_return = ThisInstrAddr(64);
    vect_offset = 0x100;

    except = ExceptionSyndrome(Exception_FIQ);

    AArch64.TakeException(EL1, except, preferred_exception_return, vect_offset);
```

Library pseudocode for aarch64/exceptions/async/AArch64.TakeVirtualIRQException

```
// AArch64.TakeVirtualIRQException()
// =================================

AArch64.TakeVirtualIRQException()
    assert PSTATE.EL IN {EL0, EL1} && EL2Enabled();
    assert HCR_EL2.TGE == '0' && HCR_EL2.IMO == '1';  // Virtual IRQ enabled if TGE==0 and IMO==1

    bits(64) preferred_exception_return = ThisInstrAddr(64);
    vect_offset = 0x80;

    except = ExceptionSyndrome(Exception_IRQ);

    AArch64.TakeException(EL1, except, preferred_exception_return, vect_offset);
```

Library pseudocode for aarch64/exceptions/async/AArch64.TakeVirtualSErrorException

```
// AArch64.TakeVirtualSErrorException()
// ====================================

AArch64.TakeVirtualSErrorException()

    assert PSTATE.EL IN {EL0, EL1} && EL2Enabled();
    assert HCR_EL2.TGE == '0' && HCR_EL2.AMO == '1';  // Virtual SError enabled if TGE==0 and AMO==1

    bits(64) preferred_exception_return = ThisInstrAddr(64);
    vect_offset = 0x180;
    except = ExceptionSyndrome(Exception_SError);

    if HaveRASExt() then
        except.syndrome<24>   = VSESR_EL2.IDS;
        except.syndrome<23:0> = VSESR_EL2.ISS;
    else
        bits(25) syndrome = bits(25) IMPLEMENTATION_DEFINED "Virtual SError syndrome";
        impdef_syndrome = syndrome<24> == '1';
        if impdef_syndrome then except.syndrome = syndrome;

    ClearPendingVirtualSError();
    AArch64.TakeException(EL1, except, preferred_exception_return, vect_offset);
```

Library pseudocode for aarch64/exceptions/debug/AArch64.BreakpointException

```
// AArch64.BreakpointException()
// =============================

AArch64.BreakpointException(FaultRecord fault)
    assert PSTATE.EL != EL3;

    route_to_el2 = (PSTATE.EL IN {EL0, EL1} && EL2Enabled() &&
                    (HCR_EL2.TGE == '1' || MDCR_EL2.TDE == '1'));

    bits(64) preferred_exception_return = ThisInstrAddr(64);
    bits(2) target_el;
    vect_offset = 0x0;
    target_el = if (PSTATE.EL == EL2 || route_to_el2) then EL2 else EL1;

    vaddress = bits(64) UNKNOWN;
    except = AArch64.AbortSyndrome(Exception_Breakpoint, fault, vaddress, target_el);
    AArch64.TakeException(target_el, except, preferred_exception_return, vect_offset);
```

Library pseudocode for aarch64/exceptions/debug/AArch64.SoftwareBreakpoint

```
// AArch64.SoftwareBreakpoint()
// ============================

AArch64.SoftwareBreakpoint(bits(16) immediate)

    route_to_el2 = (PSTATE.EL IN {EL0, EL1} &&
                    EL2Enabled() && (HCR_EL2.TGE == '1' || MDCR_EL2.TDE == '1'));

    bits(64) preferred_exception_return = ThisInstrAddr(64);
    vect_offset = 0x0;

    except = ExceptionSyndrome(Exception_SoftwareBreakpoint);
    except.syndrome<15:0> = immediate;

    if UInt(PSTATE.EL) > UInt(EL1) then
        AArch64.TakeException(PSTATE.EL, except, preferred_exception_return, vect_offset);
    elsif route_to_el2 then
        AArch64.TakeException(EL2, except, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(EL1, except, preferred_exception_return, vect_offset);
```

Library pseudocode for aarch64/exceptions/debug/AArch64.SoftwareStepException

```
// AArch64.SoftwareStepException()
// ===============================

AArch64.SoftwareStepException()
    assert PSTATE.EL != EL3;

    route_to_el2 = (PSTATE.EL IN {EL0, EL1} && EL2Enabled() &&
                    (HCR_EL2.TGE == '1' || MDCR_EL2.TDE == '1'));

    bits(64) preferred_exception_return = ThisInstrAddr(64);
    vect_offset = 0x0;

    except = ExceptionSyndrome(Exception_SoftwareStep);
    if SoftwareStep_DidNotStep() then
        except.syndrome<24> = '0';
    else
        except.syndrome<24> = '1';
        except.syndrome<6> = if SoftwareStep_SteppedEX() then '1' else '0';
    except.syndrome<5:0> = '100010';                // IFSC = Debug Exception

    if PSTATE.EL == EL2 || route_to_el2 then
        AArch64.TakeException(EL2, except, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(EL1, except, preferred_exception_return, vect_offset);
```

Library pseudocode for aarch64/exceptions/debug/AArch64.VectorCatchException

```
// AArch64.VectorCatchException()
// ==============================
// Vector Catch taken from EL0 or EL1 to EL2. This can only be called when debug exceptions are
// being routed to EL2, as Vector Catch is a legacy debug event.

AArch64.VectorCatchException(FaultRecord fault)
    assert PSTATE.EL != EL2;
    assert EL2Enabled() && (HCR_EL2.TGE == '1' || MDCR_EL2.TDE == '1');

    bits(64) preferred_exception_return = ThisInstrAddr(64);
    vect_offset = 0x0;

    vaddress = bits(64) UNKNOWN;
    except = AArch64.AbortSyndrome(Exception_VectorCatch, fault, vaddress, EL2);

    AArch64.TakeException(EL2, except, preferred_exception_return, vect_offset);
```

Library pseudocode for aarch64/exceptions/debug/AArch64.WatchpointException

```
// AArch64.WatchpointException()
// =============================

AArch64.WatchpointException(bits(64) vaddress, FaultRecord fault)
    assert PSTATE.EL != EL3;

    route_to_el2 = (PSTATE.EL IN {EL0, EL1} && EL2Enabled() &&
                    (HCR_EL2.TGE == '1' || MDCR_EL2.TDE == '1'));

    bits(64) preferred_exception_return = ThisInstrAddr(64);
    bits(2) target_el;
    vect_offset = 0x0;
    target_el = if (PSTATE.EL == EL2 || route_to_el2) then EL2 else EL1;

    ExceptionRecord except;
    if HaveNV2Ext() && fault.access.acctype == AccessType_NV2 then
        except = AArch64.AbortSyndrome(Exception_NV2Watchpoint, fault, vaddress, target_el);
    else
        except = AArch64.AbortSyndrome(Exception_Watchpoint, fault, vaddress, target_el);
    AArch64.TakeException(target_el, except, preferred_exception_return, vect_offset);
```

Library pseudocode for aarch64/exceptions/exceptions/AArch64.ExceptionClass

```
// AArch64.ExceptionClass()
// ========================
// Returns the Exception Class and Instruction Length fields to be reported in ESR

(integer,bit) AArch64.ExceptionClass(Exception exceptype, bits(2) target_el)

    il_is_valid = TRUE;
    from_32 = UsingAArch32();
    integer ec;
    case exceptype of
        when Exception_Uncategorized         ec = 0x00; il_is_valid = FALSE;
        when Exception_WFxTrap               ec = 0x01;
        when Exception_CP15RTTrap            ec = 0x03; assert from_32;
        when Exception_CP15RRTTrap           ec = 0x04; assert from_32;
        when Exception_CP14RTTrap            ec = 0x05; assert from_32;
        when Exception_CP14DTTrap            ec = 0x06; assert from_32;
        when Exception_AdvSIMDFPAccessTrap   ec = 0x07;
        when Exception_FPIDTrap              ec = 0x08;
        when Exception_PACTrap               ec = 0x09;
        when Exception_LDST64BTrap           ec = 0x0A;
        when Exception_TSTARTAccessTrap      ec = 0x1B;
        when Exception_GPC                   ec = 0x1E;
        when Exception_CP14RRTTrap           ec = 0x0C; assert from_32;
        when Exception_BranchTarget          ec = 0x0D;
        when Exception_IllegalState          ec = 0x0E; il_is_valid = FALSE;
        when Exception_SupervisorCall        ec = 0x11;
        when Exception_HypervisorCall        ec = 0x12;
        when Exception_MonitorCall           ec = 0x13;
        when Exception_SystemRegisterTrap    ec = 0x18; assert !from_32;
        when Exception_SystemRegister128Trap ec = 0x14; assert !from_32;
        when Exception_SVEAccessTrap         ec = 0x19; assert !from_32;
        when Exception_ERetTrap              ec = 0x1A; assert !from_32;
        when Exception_PACFail               ec = 0x1C; assert !from_32;
        when Exception_SMEAccessTrap         ec = 0x1D; assert !from_32;
        when Exception_InstructionAbort      ec = 0x20; il_is_valid = FALSE;
        when Exception_PCAlignment           ec = 0x22; il_is_valid = FALSE;
        when Exception_DataAbort             ec = 0x24;
        when Exception_NV2DataAbort          ec = 0x25;
        when Exception_SPAlignment           ec = 0x26; il_is_valid = FALSE; assert !from_32;
        when Exception_MemCpyMemSet          ec = 0x27;
        when Exception_GCSFail               ec = 0x2D; assert !from_32;
        when Exception_FPTrappedException    ec = 0x28;
        when Exception_SError                ec = 0x2F; il_is_valid = FALSE;
        when Exception_Breakpoint            ec = 0x30; il_is_valid = FALSE;
        when Exception_SoftwareStep          ec = 0x32; il_is_valid = FALSE;
        when Exception_Watchpoint            ec = 0x34; il_is_valid = FALSE;
        when Exception_NV2Watchpoint         ec = 0x35; il_is_valid = FALSE;
        when Exception_SoftwareBreakpoint    ec = 0x38;
        when Exception_VectorCatch           ec = 0x3A; il_is_valid = FALSE; assert from_32;
        otherwise                            Unreachable();

    if ec IN {0x20,0x24,0x30,0x32,0x34} && target_el == PSTATE.EL then
        ec = ec + 1;

    if ec IN {0x11,0x12,0x13,0x28,0x38} && !from_32 then
        ec = ec + 4;
    bit il;
    if il_is_valid then
        il = if ThisInstrLength() == 32 then '1' else '0';
    else
        il = '1';
    assert from_32 || il == '1';            // AArch64 instructions always 32-bit

    return (ec,il);
```

Library pseudocode for aarch64/exceptions/exceptions/AArch64.ReportException

```
// AArch64.ReportException()
// =========================
// Report syndrome information for exception taken to AArch64 state.

AArch64.ReportException(ExceptionRecord except, bits(2) target_el)

    Exception exceptype = except.exceptype;

    (ec,il) = AArch64.ExceptionClass(exceptype, target_el);
    iss  = except.syndrome;
    iss2 = except.syndrome2;

    // IL is not valid for Data Abort exceptions without valid instruction syndrome information
    if ec IN {0x24,0x25} && iss<24> == '0' then
        il = '1';

    ESR[target_el] = (Zeros(8)  :   // <63:56>
                      iss2      :   // <55:32>
                      ec<5:0>   :   // <31:26>
                      il        :   // <25>
                      iss);         // <24:0>

    if exceptype IN {
        Exception_InstructionAbort,
        Exception_PCAlignment,
        Exception_DataAbort,
        Exception_NV2DataAbort,
        Exception_NV2Watchpoint,
        Exception_GPC,
        Exception_Watchpoint
    } then
        FAR[target_el] = except.vaddress;
    else
        FAR[target_el] = bits(64) UNKNOWN;

    if except.ipavalid then
        HPFAR_EL2<47:4> = except.ipaddress<55:12>;
        if IsSecureEL2Enabled() && CurrentSecurityState() == SS_Secure then
            HPFAR_EL2.NS = except.NS;
        else
            HPFAR_EL2.NS = '0';
    elsif target_el == EL2 then
        HPFAR_EL2<47:4> = bits(44) UNKNOWN;

    if except.pavalid then
        bits(64) faultaddr = ZeroExtend(except.paddress.address, 64);
        if HaveRME() then
            case except.paddress.paspace of
                when PAS_Secure     faultaddr<63:62> = '00';
                when PAS_NonSecure  faultaddr<63:62> = '10';
                when PAS_Root       faultaddr<63:62> = '01';
                when PAS_Realm      faultaddr<63:62> = '11';
            if exceptype == Exception_GPC then
                faultaddr<11:0> = Zeros(12);
        else
            faultaddr<63> = if except.paddress.paspace == PAS_NonSecure then '1' else '0';
        PFAR[target_el] = faultaddr;
    elsif HavePFAR() || (HaveRME() && target_el == EL3) then
        PFAR[target_el] = bits(64) UNKNOWN;
    return;
```

Library pseudocode for aarch64/exceptions/exceptions/AArch64.ResetControlRegisters

```
// AArch64.ResetControlRegisters()
// ===============================
// Resets System registers and memory-mapped control registers that have architecturally-defined
// reset values to those values.

AArch64.ResetControlRegisters(boolean cold_reset);
```

Library pseudocode for aarch64/exceptions/exceptions/AArch64.TakeReset

```
// AArch64.TakeReset()
// ===================
// Reset into AArch64 state

AArch64.TakeReset(boolean cold_reset)
    assert HaveAArch64();

    // Enter the highest implemented Exception level in AArch64 state
    PSTATE.nRW = '0';
    if HaveEL(EL3) then
        PSTATE.EL = EL3;
    elsif HaveEL(EL2) then
        PSTATE.EL = EL2;
    else
        PSTATE.EL = EL1;

    // Reset System registers
    // and other system components
    AArch64.ResetControlRegisters(cold_reset);

    // Reset all other PSTATE fields
    PSTATE.SP = '1';                    // Select stack pointer
    PSTATE.<D,A,I,F>  = '1111';         // All asynchronous exceptions masked
    PSTATE.SS = '0';                    // Clear software step bit
    PSTATE.DIT = '0';                   // PSTATE.DIT is reset to 0 when resetting into AArch64
    PSTATE.IL = '0';                    // Clear Illegal Execution state bit

    if HaveTME() then TSTATE.depth = 0;             // Non-transactional state

    // All registers, bits and fields not reset by the above pseudocode or by the BranchTo() call
    // below are UNKNOWN bitstrings after reset. In particular, the return information registers
    // ELR_ELx and SPSR_ELx have UNKNOWN values, so that it
    // is impossible to return from a reset in an architecturally defined way.
    AArch64.ResetGeneralRegisters();
    AArch64.ResetSIMDFPRegisters();
    AArch64.ResetSpecialRegisters();
    ResetExternalDebugRegisters(cold_reset);

    bits(64) rv;                      // IMPLEMENTATION DEFINED reset vector

    if HaveEL(EL3) then
        rv = RVBAR_EL3;
    elsif HaveEL(EL2) then
        rv = RVBAR_EL2;
    else
        rv = RVBAR_EL1;

    // The reset vector must be correctly aligned
    assert IsZero(rv<63:AArch64.PAMax()>) && IsZero(rv<1:0>);

    boolean branch_conditional = FALSE;
    BranchTo(rv, BranchType_RESET, branch_conditional);
```

Library pseudocode for aarch64/exceptions/ieeefp/AArch64.FPTrappedException

```
// AArch64.FPTrappedException()
// ============================

AArch64.FPTrappedException(boolean is_ase, bits(8) accumulated_exceptions)
    except = ExceptionSyndrome(Exception_FPTrappedException);
    if is_ase then
        if boolean IMPLEMENTATION_DEFINED "vector instructions set TFV to 1" then
            except.syndrome<23> = '1';                          // TFV
        else
            except.syndrome<23> = '0';                          // TFV
    else
        except.syndrome<23> = '1';                              // TFV
    except.syndrome<10:8> = bits(3) UNKNOWN;                    // VECITR
    if except.syndrome<23> == '1' then
        except.syndrome<7,4:0> = accumulated_exceptions<7,4:0>; // IDF,IXF,UFF,OFF,DZF,IOF
    else
        except.syndrome<7,4:0> = bits(6) UNKNOWN;

    route_to_el2 = EL2Enabled() && HCR_EL2.TGE == '1';

    bits(64) preferred_exception_return = ThisInstrAddr(64);
    vect_offset = 0x0;

    if UInt(PSTATE.EL) > UInt(EL1) then
        AArch64.TakeException(PSTATE.EL, except, preferred_exception_return, vect_offset);
    elsif route_to_el2 then
        AArch64.TakeException(EL2, except, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(EL1, except, preferred_exception_return, vect_offset);
```

Library pseudocode for aarch64/exceptions/syscalls/AArch64.CallHypervisor

```
// AArch64.CallHypervisor()
// ========================
// Performs a HVC call

AArch64.CallHypervisor(bits(16) immediate)
    assert HaveEL(EL2);

    if UsingAArch32() then AArch32.ITAdvance();
    SSAdvance();
    bits(64) preferred_exception_return = NextInstrAddr(64);
    vect_offset = 0x0;

    except = ExceptionSyndrome(Exception_HypervisorCall);
    except.syndrome<15:0> = immediate;

    if PSTATE.EL == EL3 then
        AArch64.TakeException(EL3, except, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(EL2, except, preferred_exception_return, vect_offset);
```

Library pseudocode for aarch64/exceptions/syscalls/AArch64.CallSecureMonitor

```
// AArch64.CallSecureMonitor()
// ===========================

AArch64.CallSecureMonitor(bits(16) immediate)
    assert HaveEL(EL3) && !ELUsingAArch32(EL3);
    if UsingAArch32() then AArch32.ITAdvance();
    SSAdvance();
    bits(64) preferred_exception_return = NextInstrAddr(64);
    vect_offset = 0x0;

    except = ExceptionSyndrome(Exception_MonitorCall);
    except.syndrome<15:0> = immediate;

    AArch64.TakeException(EL3, except, preferred_exception_return, vect_offset);
```

Library pseudocode for aarch64/exceptions/syscalls/AArch64.CallSupervisor

```
// AArch64.CallSupervisor()
// ========================
// Calls the Supervisor

AArch64.CallSupervisor(bits(16) immediate_in)
    bits(16) immediate = immediate_in;
    if UsingAArch32() then AArch32.ITAdvance();
    SSAdvance();
    route_to_el2 = PSTATE.EL == EL0 && EL2Enabled() && HCR_EL2.TGE == '1';

    bits(64) preferred_exception_return = NextInstrAddr(64);
    vect_offset = 0x0;

    except = ExceptionSyndrome(Exception_SupervisorCall);
    except.syndrome<15:0> = immediate;

    if UInt(PSTATE.EL) > UInt(EL1) then
        AArch64.TakeException(PSTATE.EL, except, preferred_exception_return, vect_offset);
    elsif route_to_el2 then
        AArch64.TakeException(EL2, except, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(EL1, except, preferred_exception_return, vect_offset);
```

Library pseudocode for aarch64/exceptions/takeexception/AArch64.TakeException

```
// AArch64.TakeException()
// =======================
// Take an exception to an Exception level using AArch64.

AArch64.TakeException(bits(2) target_el, ExceptionRecord exception_in,
                      bits(64) preferred_exception_return, integer vect_offset_in)
    assert HaveEL(target_el) && !ELUsingAArch32(target_el) && UInt(target_el) >= UInt(PSTATE.EL);
    if Halted() then
        AArch64.TakeExceptionInDebugState(target_el, exception_in);
        return;
    ExceptionRecord except = exception_in;
    boolean sync_errors;
    boolean iesb_req;
    if HaveIESB() then
        sync_errors = SCTLR[target_el].IESB == '1';
        if HaveDoubleFaultExt() then
            sync_errors = sync_errors || (SCR_EL3.<EA,NMEA> == '11' && target_el == EL3);
        if sync_errors && InsertIESBBeforeException(target_el) then
            SynchronizeErrors();
            iesb_req = FALSE;
            sync_errors = FALSE;
            TakeUnmaskedPhysicalSErrorInterrupts(iesb_req);
    else
        sync_errors = FALSE;

    if HaveTME() && TSTATE.depth > 0 then
        TMFailure cause;
        case except.exceptype of
            when Exception_SoftwareBreakpoint cause = TMFailure_DBG;
            when Exception_Breakpoint         cause = TMFailure_DBG;
            when Exception_Watchpoint         cause = TMFailure_DBG;
            when Exception_SoftwareStep       cause = TMFailure_DBG;
            otherwise                         cause = TMFailure_ERR;
        FailTransaction(cause, FALSE);

    SynchronizeContext();

    // If coming from AArch32 state, the top parts of the X[] registers might be set to zero
    from_32 = UsingAArch32();
    if from_32 then AArch64.MaybeZeroRegisterUppers();
    if from_32 && HaveSME() && PSTATE.SM == '1' then
        ResetSVEState();
    else
        MaybeZeroSVEUppers(target_el);

    integer vect_offset = vect_offset_in;
    if UInt(target_el) > UInt(PSTATE.EL) then
        boolean lower_32;
        if target_el == EL3 then
            if EL2Enabled() then
                lower_32 = ELUsingAArch32(EL2);
            else
                lower_32 = ELUsingAArch32(EL1);
        elsif IsInHost() && PSTATE.EL == EL0 && target_el == EL2 then
            lower_32 = ELUsingAArch32(EL0);
        else
            lower_32 = ELUsingAArch32(target_el - 1);
        vect_offset = vect_offset + (if lower_32 then 0x600 else 0x400);

    elsif PSTATE.SP == '1' then
        vect_offset = vect_offset + 0x200;

    bits(64) spsr = GetPSRFromPSTATE(AArch64_NonDebugState, 64);

    if PSTATE.EL == EL1 && target_el == EL1 && EL2Enabled() then
        if HaveNV2Ext() && (HCR_EL2.<NV,NV1,NV2> == '100' || HCR_EL2.<NV,NV1,NV2> == '111') then
            spsr<3:2> = '10';
        else
            if HaveNVExt() && HCR_EL2.<NV,NV1> == '10' then
                spsr<3:2> = '10';
```

```
if HaveBTIExt() && !UsingAArch32() then
    boolean zero_btype;
    // SPSR[].BTYPE is only guaranteed valid for these exception types
    if except.exceptype IN {Exception_SError, Exception_IRQ, Exception_FIQ,
                            Exception_SoftwareStep, Exception_PCAlignment,
                            Exception_InstructionAbort, Exception_Breakpoint,
                            Exception_VectorCatch, Exception_SoftwareBreakpoint,
                            Exception_IllegalState, Exception_BranchTarget} then
        zero_btype = FALSE;
    else
        zero_btype = ConstrainUnpredictableBool(Unpredictable_ZEROBTYPE);
    if zero_btype then spsr<11:10> = '00';

if HaveNV2Ext() && except.exceptype == Exception_NV2DataAbort && target_el == EL3 then
    // External aborts are configured to be taken to EL3
    except.exceptype = Exception_DataAbort;
if !(except.exceptype IN {Exception_IRQ, Exception_FIQ}) then
    AArch64.ReportException(except, target_el);

if HaveBRBExt() then
    BRBEException(except, preferred_exception_return,
                  VBAR[target_el]<63:11>:vect_offset<10:0>, target_el,
                  except.trappedsyscallinst);

if HaveGCS() then
    if PSTATE.EL == target_el then
        if GetCurrentEXLOCKEN() then
            PSTATE.EXLOCK = '1';
        else
            PSTATE.EXLOCK = '0';
    else
        PSTATE.EXLOCK = '0';

PSTATE.EL = target_el;
PSTATE.nRW = '0';
PSTATE.SP = '1';

SPSR[] = spsr;
ELR[] = preferred_exception_return;

PSTATE.SS = '0';
if HaveFeatNMI() && !ELUsingAArch32(target_el) then PSTATE.ALLINT = NOT SCTLR[].SPINTMASK;
PSTATE.<D,A,I,F> = '1111';
PSTATE.IL = '0';
if from_32 then                             // Coming from AArch32
    PSTATE.IT = '00000000';
    PSTATE.T = '0';                         // PSTATE.J is RES0
if (HavePANExt() && (PSTATE.EL == EL1 || (PSTATE.EL == EL2 && ELIsInHost(EL0))) &&
    SCTLR[].SPAN == '0') then
    PSTATE.PAN = '1';
if HaveUAOExt() then PSTATE.UAO = '0';
if HaveBTIExt() then PSTATE.BTYPE = '00';
if HaveSSBSExt() then PSTATE.SSBS = SCTLR[].DSSBS;
if HaveMTEExt() then PSTATE.TCO = '1';

boolean branch_conditional = FALSE;
BranchTo(VBAR[]<63:11>:vect_offset<10:0>, BranchType_EXCEPTION, branch_conditional);

CheckExceptionCatch(TRUE);                  // Check for debug event on exception entry

if sync_errors then
    SynchronizeErrors();
    iesb_req = TRUE;
    TakeUnmaskedPhysicalSErrorInterrupts(iesb_req);

EndOfInstruction();
```

Library pseudocode for aarch64/exceptions/traps/AArch64.AArch32SystemAccessTrap

```
// AArch64.AArch32SystemAccessTrap()
// =================================
// Trapped AARCH32 System register access.

AArch64.AArch32SystemAccessTrap(bits(2) target_el, integer ec)
    assert HaveEL(target_el) && target_el != EL0 && UInt(target_el) >= UInt(PSTATE.EL);

    bits(64) preferred_exception_return = ThisInstrAddr(64);
    vect_offset = 0x0;

    except = AArch64.AArch32SystemAccessTrapSyndrome(ThisInstr(), ec);
    AArch64.TakeException(target_el, except, preferred_exception_return, vect_offset);
```

Library pseudocode for aarch64/exceptions/traps/AArch64.AArch32SystemAccessTrapSyndrome

```
// AArch64.AArch32SystemAccessTrapSyndrome()
// =========================================
// Returns the syndrome information for traps on AArch32 MCR, MCRR, MRC, MRRC, and VMRS,
// VMSR instructions, other than traps that are due to HCPTR or CPACR.

ExceptionRecord AArch64.AArch32SystemAccessTrapSyndrome(bits(32) instr, integer ec)
    ExceptionRecord except;

    case ec of
        when 0x0    except = ExceptionSyndrome(Exception_Uncategorized);
        when 0x3    except = ExceptionSyndrome(Exception_CP15RTTrap);
        when 0x4    except = ExceptionSyndrome(Exception_CP15RRTTrap);
        when 0x5    except = ExceptionSyndrome(Exception_CP14RTTrap);
        when 0x6    except = ExceptionSyndrome(Exception_CP14DTTrap);
        when 0x7    except = ExceptionSyndrome(Exception_AdvSIMDFPAccessTrap);
        when 0x8    except = ExceptionSyndrome(Exception_FPIDTrap);
        when 0xC    except = ExceptionSyndrome(Exception_CP14RRTTrap);
        otherwise   Unreachable();

    bits(20) iss = Zeros(20);

    if except.exceptype == Exception_Uncategorized then
        return except;
    elsif except.exceptype IN {Exception_FPIDTrap, Exception_CP14RTTrap,
                               Exception_CP15RTTrap} then
        // Trapped MRC/MCR, VMRS on FPSID
        if except.exceptype != Exception_FPIDTrap then    // When trap is not for VMRS
            iss<19:17> = instr<7:5>;           // opc2
            iss<16:14> = instr<23:21>;         // opc1
            iss<13:10> = instr<19:16>;         // CRn
            iss<4:1>   = instr<3:0>;           // CRm
        else
            iss<19:17> = '000';
            iss<16:14> = '111';
            iss<13:10> = instr<19:16>;         // reg
            iss<4:1>   = '0000';

        if instr<20> == '1' && instr<15:12> == '1111' then    // MRC, Rt==15
            iss<9:5> = '11111';
        elsif instr<20> == '0' && instr<15:12> == '1111' then // MCR, Rt==15
            iss<9:5> = bits(5) UNKNOWN;
        else
            iss<9:5> = LookUpRIndex(UInt(instr<15:12>), PSTATE.M)<4:0>;
    elsif except.exceptype IN {Exception_CP14RRTTrap, Exception_AdvSIMDFPAccessTrap,
                            Exception_CP15RRTTrap} then
        // Trapped MRRC/MCRR, VMRS/VMSR
        iss<19:16> = instr<7:4>;          // opc1
        if instr<19:16> == '1111' then    // Rt2==15
            iss<14:10> = bits(5) UNKNOWN;
        else
            iss<14:10> = LookUpRIndex(UInt(instr<19:16>), PSTATE.M)<4:0>;

        if instr<15:12> == '1111' then    // Rt==15
            iss<9:5> = bits(5) UNKNOWN;
        else
            iss<9:5> = LookUpRIndex(UInt(instr<15:12>), PSTATE.M)<4:0>;
        iss<4:1>   = instr<3:0>;         // CRm
    elsif except.exceptype == Exception_CP14DTTrap then
        // Trapped LDC/STC
        iss<19:12> = instr<7:0>;         // imm8
        iss<4>     = instr<23>;          // U
        iss<2:1>   = instr<24,21>;       // P,W
        if instr<19:16> == '1111' then   // Rn==15, LDC(Literal addressing)/STC
            iss<9:5> = bits(5) UNKNOWN;
            iss<3>   = '1';
    iss<0> = instr<20>;                  // Direction

    except.syndrome<24:20> = ConditionSyndrome();
    except.syndrome<19:0>  = iss;
```

```
    return except;
```

Library pseudocode for aarch64/exceptions/traps/AArch64.AdvSIMDFPAccessTrap

```
// AArch64.AdvSIMDFPAccessTrap()
// =============================
// Trapped access to Advanced SIMD or FP registers due to CPACR[].

AArch64.AdvSIMDFPAccessTrap(bits(2) target_el)
    bits(64) preferred_exception_return = ThisInstrAddr(64);
    ExceptionRecord except;
    vect_offset = 0x0;

    route_to_el2 = (target_el == EL1 && EL2Enabled() && HCR_EL2.TGE == '1');

    if route_to_el2 then
        except = ExceptionSyndrome(Exception_Uncategorized);
        AArch64.TakeException(EL2, except, preferred_exception_return, vect_offset);
    else
        except = ExceptionSyndrome(Exception_AdvSIMDFPAccessTrap);
        except.syndrome<24:20> = ConditionSyndrome();
        AArch64.TakeException(target_el, except, preferred_exception_return, vect_offset);

    return;
```

Library pseudocode for aarch64/exceptions/traps/AArch64.CheckCP15InstrCoarseTraps

```
// AArch64.CheckCP15InstrCoarseTraps()
// ===================================
// Check for coarse-grained AArch32  traps to System registers in the
// coproc=0b1111 encoding space by HSTR_EL2, HCR_EL2, and SCTLR_ELx.

AArch64.CheckCP15InstrCoarseTraps(integer CRn, integer nreg, integer CRm)
    trapped_encoding = ((CRn == 9  && CRm IN {0,1,2,    5,6,7,8   }) ||
                        (CRn == 10 && CRm IN {0,1,    4,      8   }) ||
                        (CRn == 11 && CRm IN {0,1,2,3,4,5,6,7,8,15}));

    // Check for MRC and MCR disabled by SCTLR_EL1.TIDCP.
    if (HaveFeatTIDCP1() && PSTATE.EL == EL0 && !IsInHost() &&
            !ELUsingAArch32(EL1) && SCTLR_EL1.TIDCP == '1' && trapped_encoding) then
        if EL2Enabled() && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x3);
        else
            AArch64.AArch32SystemAccessTrap(EL1, 0x3);

    // Check for coarse-grained Hyp traps
    if PSTATE.EL IN {EL0, EL1} && EL2Enabled() then
        // Check for MRC and MCR disabled by SCTLR_EL2.TIDCP.
        if (HaveFeatTIDCP1() && PSTATE.EL == EL0 && IsInHost() &&
                SCTLR_EL2.TIDCP == '1' && trapped_encoding) then
            AArch64.AArch32SystemAccessTrap(EL2, 0x3);

        major = if nreg == 1 then CRn else CRm;
        // Check for MCR, MRC, MCRR, and MRRC disabled by HSTR_EL2<CRn/CRm>
        // and MRC and MCR disabled by HCR_EL2.TIDCP.
        if ((!IsInHost() && !(major IN {4,14}) && HSTR_EL2<major> == '1') ||
                (HCR_EL2.TIDCP == '1' && nreg == 1 && trapped_encoding)) then
            if (PSTATE.EL == EL0 &&
                    boolean IMPLEMENTATION_DEFINED "UNDEF unallocated CP15 access at EL0") then
                UNDEFINED;
            AArch64.AArch32SystemAccessTrap(EL2, 0x3);
```

Library pseudocode for aarch64/exceptions/traps/AArch64.CheckFPAdvSIMDEnabled

```
// AArch64.CheckFPAdvSIMDEnabled()
// ===============================

AArch64.CheckFPAdvSIMDEnabled()
    AArch64.CheckFPEnabled();
    // Check for illegal use of Advanced
    // SIMD in Streaming SVE Mode
    if HaveSME() && PSTATE.SM == '1' && !IsFullA64Enabled() then
        SMEAccessTrap(SMEExceptionType_Streaming, PSTATE.EL);
```

Library pseudocode for aarch64/exceptions/traps/AArch64.CheckFPAdvSIMDTrap

```
// AArch64.CheckFPAdvSIMDTrap()
// ============================
// Check against CPTR_EL2 and CPTR_EL3.

AArch64.CheckFPAdvSIMDTrap()
    if HaveEL(EL3) && CPTR_EL3.TFP == '1' && EL3SDDUndefPriority() then
        UNDEFINED;

    if PSTATE.EL IN {EL0, EL1, EL2} && EL2Enabled() then
        // Check if access disabled in CPTR_EL2
        if HaveVirtHostExt() && HCR_EL2.E2H == '1' then
            boolean disabled;
            case CPTR_EL2.FPEN of
                when 'x0' disabled = TRUE;
                when '01' disabled = PSTATE.EL == EL0 && HCR_EL2.TGE == '1';
                when '11' disabled = FALSE;
            if disabled then AArch64.AdvSIMDFPAccessTrap(EL2);
        else
            if CPTR_EL2.TFP == '1' then AArch64.AdvSIMDFPAccessTrap(EL2);

    if HaveEL(EL3) then
        // Check if access disabled in CPTR_EL3
        if CPTR_EL3.TFP == '1' then
            if EL3SDDUndef() then
                UNDEFINED;
            else
                AArch64.AdvSIMDFPAccessTrap(EL3);
```

Library pseudocode for aarch64/exceptions/traps/AArch64.CheckFPEnabled

```
// AArch64.CheckFPEnabled()
// ========================
// Check against CPACR[]

AArch64.CheckFPEnabled()
    if PSTATE.EL IN {EL0, EL1} && !IsInHost() then
        // Check if access disabled in CPACR_EL1
        boolean disabled;
        case CPACR_EL1.FPEN of
            when 'x0' disabled = TRUE;
            when '01' disabled = PSTATE.EL == EL0;
            when '11' disabled = FALSE;
        if disabled then AArch64.AdvSIMDFPAccessTrap(EL1);

    AArch64.CheckFPAdvSIMDTrap();               // Also check against CPTR_EL2 and CPTR_EL3
```

Library pseudocode for aarch64/exceptions/traps/AArch64.CheckForERetTrap

```
// AArch64.CheckForERetTrap()
// ==========================
// Check for trap on ERET, ERETAA, ERETAB instruction

AArch64.CheckForERetTrap(boolean eret_with_pac, boolean pac_uses_key_a)

    route_to_el2 = FALSE;
    // Non-secure EL1 execution of ERET, ERETAA, ERETAB when either HCR_EL2.NV or
    // HFGITR_EL2.ERET is set, is trapped to EL2
    route_to_el2 = (PSTATE.EL == EL1 && EL2Enabled() &&
                    ((HaveNVExt() && HCR_EL2.NV == '1') ||
                     (HaveFGTExt() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
                      HFGITR_EL2.ERET == '1')));
    if route_to_el2 then
        ExceptionRecord except;
        bits(64) preferred_exception_return = ThisInstrAddr(64);
        vect_offset = 0x0;
        except = ExceptionSyndrome(Exception_ERetTrap);
        if !eret_with_pac then                              // ERET
            except.syndrome<1> = '0';
            except.syndrome<0> = '0';                       // RES0
        else
            except.syndrome<1> = '1';
            if pac_uses_key_a then                          // ERETAA
                except.syndrome<0> = '0';
            else    // ERETAB
                except.syndrome<0> = '1';
        AArch64.TakeException(EL2, except, preferred_exception_return, vect_offset);
```

Library pseudocode for aarch64/exceptions/traps/AArch64.CheckForSMCUndefOrTrap

```
// AArch64.CheckForSMCUndefOrTrap()
// ================================
// Check for UNDEFINED or trap on SMC instruction

AArch64.CheckForSMCUndefOrTrap(bits(16) imm)
    if PSTATE.EL == EL0 then UNDEFINED;
    if (!(PSTATE.EL == EL1 && EL2Enabled() && HCR_EL2.TSC == '1') &&
        HaveEL(EL3) && SCR_EL3.SMD == '1') then
        UNDEFINED;
    route_to_el2 = FALSE;
    if !HaveEL(EL3) then
        if PSTATE.EL == EL1 && EL2Enabled() then
            if HaveNVExt() && HCR_EL2.NV == '1' && HCR_EL2.TSC == '1' then
                route_to_el2 = TRUE;
            else
                UNDEFINED;
        else
            UNDEFINED;
    else
        route_to_el2 = PSTATE.EL == EL1 && EL2Enabled() && HCR_EL2.TSC == '1';
    if route_to_el2 then
        bits(64) preferred_exception_return = ThisInstrAddr(64);
        vect_offset = 0x0;
        except = ExceptionSyndrome(Exception_MonitorCall);
        except.syndrome<15:0> = imm;
        except.trappedsyscallinst = TRUE;
        AArch64.TakeException(EL2, except, preferred_exception_return, vect_offset);
```

Library pseudocode for aarch64/exceptions/traps/AArch64.CheckForSVCTrap

```
// AArch64.CheckForSVCTrap()
// =========================
// Check for trap on SVC instruction

AArch64.CheckForSVCTrap(bits(16) immediate)
    if HaveFGTExt() then
        route_to_el2 = FALSE;
        if PSTATE.EL == EL0 then
            route_to_el2 = (!UsingAArch32() && !ELUsingAArch32(EL1) &&
                           EL2Enabled() && HFGITR_EL2.SVC_EL0 == '1' &&
                           (HCR_EL2.<E2H, TGE> != '11' && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1')));

        elsif PSTATE.EL == EL1 then
            route_to_el2 = (EL2Enabled() && HFGITR_EL2.SVC_EL1 == '1' &&
                           (!HaveEL(EL3) || SCR_EL3.FGTEn == '1'));

        if route_to_el2 then
            except = ExceptionSyndrome(Exception_SupervisorCall);
            except.syndrome<15:0> = immediate;
            except.trappedsyscallinst = TRUE;
            bits(64) preferred_exception_return = ThisInstrAddr(64);
            vect_offset = 0x0;

            AArch64.TakeException(EL2, except, preferred_exception_return, vect_offset);
```

Library pseudocode for aarch64/exceptions/traps/AArch64.CheckForWFxTrap

```
// AArch64.CheckForWFxTrap()
// =========================
// Check for trap on WFE or WFI instruction

AArch64.CheckForWFxTrap(bits(2) target_el, WFxType wfxtype)
    assert HaveEL(target_el);

    boolean is_wfe = wfxtype IN {WFxType_WFE, WFxType_WFET};
    boolean trap;
    case target_el of
        when EL1
            trap = (if is_wfe then SCTLR[].nTWE else SCTLR[].nTWI) == '0';
        when EL2
            trap = (if is_wfe then HCR_EL2.TWE else HCR_EL2.TWI) == '1';
        when EL3
            trap = (if is_wfe then SCR_EL3.TWE else SCR_EL3.TWI) == '1';

    if trap then
        AArch64.WFxTrap(wfxtype, target_el);
```

Library pseudocode for aarch64/exceptions/traps/AArch64.CheckIllegalState

```
// AArch64.CheckIllegalState()
// ===========================
// Check PSTATE.IL bit and generate Illegal Execution state exception if set.

AArch64.CheckIllegalState()
    if PSTATE.IL == '1' then
        route_to_el2 = PSTATE.EL == EL0 && EL2Enabled() && HCR_EL2.TGE == '1';

        bits(64) preferred_exception_return = ThisInstrAddr(64);
        vect_offset = 0x0;

        except = ExceptionSyndrome(Exception_IllegalState);

        if UInt(PSTATE.EL) > UInt(EL1) then
            AArch64.TakeException(PSTATE.EL, except, preferred_exception_return, vect_offset);
        elsif route_to_el2 then
            AArch64.TakeException(EL2, except, preferred_exception_return, vect_offset);
        else
            AArch64.TakeException(EL1, except, preferred_exception_return, vect_offset);
```

Library pseudocode for aarch64/exceptions/traps/AArch64.MonitorModeTrap

```
// AArch64.MonitorModeTrap()
// =========================
// Trapped use of Monitor mode features in a Secure EL1 AArch32 mode

AArch64.MonitorModeTrap()
    bits(64) preferred_exception_return = ThisInstrAddr(64);
    vect_offset = 0x0;

    except = ExceptionSyndrome(Exception_Uncategorized);

    if IsSecureEL2Enabled() then
        AArch64.TakeException(EL2, except, preferred_exception_return, vect_offset);
    AArch64.TakeException(EL3, except, preferred_exception_return, vect_offset);
```

Library pseudocode for aarch64/exceptions/traps/AArch64.SystemAccessTrap

```
// AArch64.SystemAccessTrap()
// ==========================
// Trapped access to AArch64 System register or system instruction.

AArch64.SystemAccessTrap(bits(2) target_el, integer ec)
    assert HaveEL(target_el) && target_el != EL0 && UInt(target_el) >= UInt(PSTATE.EL);

    bits(64) preferred_exception_return = ThisInstrAddr(64);
    vect_offset = 0x0;

    except = AArch64.SystemAccessTrapSyndrome(ThisInstr(), ec);
    AArch64.TakeException(target_el, except, preferred_exception_return, vect_offset);
```

Library pseudocode for aarch64/exceptions/traps/AArch64.SystemAccessTrapSyndrome

```
// AArch64.SystemAccessTrapSyndrome()
// ==================================
// Returns the syndrome information for traps on AArch64 MSR/MRS instructions.

ExceptionRecord AArch64.SystemAccessTrapSyndrome(bits(32) instr_in, integer ec)
    ExceptionRecord except;
    bits(32) instr = instr_in;
    case ec of
        when 0x0                         // Trapped access due to unknown reason.
            except = ExceptionSyndrome(Exception_Uncategorized);
        when 0x7                         // Trapped access to SVE, Advance SIMD&FP System register.
            except = ExceptionSyndrome(Exception_AdvSIMDFPAccessTrap);
            except.syndrome<24:20> = ConditionSyndrome();
        when 0x14                        // Trapped access to 128-bit System register or
                                         // 128-bit System instruction.
            except = ExceptionSyndrome(Exception_SystemRegister128Trap);
            instr = ThisInstr();
            except.syndrome<21:20> = instr<20:19>; // Op0
            except.syndrome<19:17> = instr<7:5>;   // Op2
            except.syndrome<16:14> = instr<18:16>; // Op1
            except.syndrome<13:10> = instr<15:12>; // CRn
            except.syndrome<9:6>   = instr<4:1>;   // Rt
            except.syndrome<4:1>   = instr<11:8>;  // CRm
            except.syndrome<0>     = instr<21>;    // Direction
        when 0x18                        // Trapped access to System register or system instruction.
            except = ExceptionSyndrome(Exception_SystemRegisterTrap);
            instr = ThisInstr();
            except.syndrome<21:20> = instr<20:19>; // Op0
            except.syndrome<19:17> = instr<7:5>;   // Op2
            except.syndrome<16:14> = instr<18:16>; // Op1
            except.syndrome<13:10> = instr<15:12>; // CRn
            except.syndrome<9:5>   = instr<4:0>;   // Rt
            except.syndrome<4:1>   = instr<11:8>;  // CRm
            except.syndrome<0>     = instr<21>;    // Direction
        when 0x19                        // Trapped access to SVE System register
            except = ExceptionSyndrome(Exception_SVEAccessTrap);
        when 0x1D                        // Trapped access to SME System register
            except = ExceptionSyndrome(Exception_SMEAccessTrap);
        otherwise
            Unreachable();

    return except;
```

Library pseudocode for aarch64/exceptions/traps/AArch64.UndefinedFault

```
// AArch64.UndefinedFault()
// ========================

AArch64.UndefinedFault()

    route_to_el2 = PSTATE.EL == EL0 && EL2Enabled() && HCR_EL2.TGE == '1';
    bits(64) preferred_exception_return = ThisInstrAddr(64);
    vect_offset = 0x0;

    except = ExceptionSyndrome(Exception_Uncategorized);

    if UInt(PSTATE.EL) > UInt(EL1) then
        AArch64.TakeException(PSTATE.EL, except, preferred_exception_return, vect_offset);
    elsif route_to_el2 then
        AArch64.TakeException(EL2, except, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(EL1, except, preferred_exception_return, vect_offset);
```

Library pseudocode for aarch64/exceptions/traps/AArch64.WFxTrap

```
// AArch64.WFxTrap()
// =================

AArch64.WFxTrap(WFxType wfxtype, bits(2) target_el)
    assert UInt(target_el) > UInt(PSTATE.EL);

    bits(64) preferred_exception_return = ThisInstrAddr(64);
    vect_offset = 0x0;

    except = ExceptionSyndrome(Exception_WFxTrap);
    except.syndrome<24:20> = ConditionSyndrome();

    case wfxtype of
        when WFxType_WFI
            except.syndrome<1:0> = '00';
        when WFxType_WFE
            except.syndrome<1:0> = '01';
        when WFxType_WFIT
            except.syndrome<1:0> = '10';
            except.syndrome<2>   = '1';   // Register field is valid
            except.syndrome<9:5> = ThisInstr()<4:0>;
        when WFxType_WFET
            except.syndrome<1:0> = '11';
            except.syndrome<2>   = '1';   // Register field is valid
            except.syndrome<9:5> = ThisInstr()<4:0>;

    if target_el == EL1 && EL2Enabled() && HCR_EL2.TGE == '1' then
        AArch64.TakeException(EL2, except, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(target_el, except, preferred_exception_return, vect_offset);
```

Library pseudocode for aarch64/exceptions/traps/CheckFPAdvSIMDEnabled64

```
// CheckFPAdvSIMDEnabled64()
// =========================
// AArch64 instruction wrapper

CheckFPAdvSIMDEnabled64()
    AArch64.CheckFPAdvSIMDEnabled();
```

Library pseudocode for aarch64/exceptions/traps/CheckFPEnabled64

```
// CheckFPEnabled64()
// ==================
// AArch64 instruction wrapper

CheckFPEnabled64()
    AArch64.CheckFPEnabled();
```

Library pseudocode for aarch64/exceptions/traps/CheckLDST64BEnabled

```
// CheckLDST64BEnabled()
// =====================
// Checks for trap on ST64B and LD64B instructions

CheckLDST64BEnabled()
    boolean trap = FALSE;
    bits(25) iss = ZeroExtend('10', 25);  // 0x2
    bits(2) target_el;

    if PSTATE.EL == EL0 then
        if !IsInHost() then
            trap = SCTLR_EL1.EnALS == '0';
            target_el = if EL2Enabled() && HCR_EL2.TGE == '1' then EL2 else EL1;
        else
            trap = SCTLR_EL2.EnALS == '0';
            target_el = EL2;
    else
        target_el = EL1;

    if (!trap && EL2Enabled() &&
        ((PSTATE.EL == EL0 && !IsInHost()) || PSTATE.EL == EL1)) then
        trap = !IsHCRXEL2Enabled() || HCRX_EL2.EnALS == '0';
        target_el = EL2;

    if trap then LDST64BTrap(target_el, iss);
```

Library pseudocode for aarch64/exceptions/traps/CheckST64BV0Enabled

```
// CheckST64BV0Enabled()
// =====================
// Checks for trap on ST64BV0 instruction

CheckST64BV0Enabled()
    boolean trap = FALSE;
    bits(25) iss = ZeroExtend('1', 25);  // 0x1
    bits(2) target_el;

    if (PSTATE.EL != EL3 && HaveEL(EL3) &&
        SCR_EL3.EnAS0 == '0' && EL3SDDUndefPriority()) then
        UNDEFINED;

    if PSTATE.EL == EL0 then
        if !IsInHost() then
            trap = SCTLR_EL1.EnAS0 == '0';
            target_el = if EL2Enabled() && HCR_EL2.TGE == '1' then EL2 else EL1;
        else
            trap = SCTLR_EL2.EnAS0 == '0';
            target_el = EL2;

    if (!trap && EL2Enabled() &&
        ((PSTATE.EL == EL0 && !IsInHost()) || PSTATE.EL == EL1)) then
        trap = !IsHCRXEL2Enabled() || HCRX_EL2.EnAS0 == '0';
        target_el = EL2;

    if !trap && PSTATE.EL != EL3 then
        trap = HaveEL(EL3) && SCR_EL3.EnAS0 == '0';
        target_el = EL3;

    if trap then
        if target_el == EL3 && EL3SDDUndef() then
            UNDEFINED;
        else
            LDST64BTrap(target_el, iss);
```

Library pseudocode for aarch64/exceptions/traps/CheckST64BVEnabled

```
// CheckST64BVEnabled()
// ====================
// Checks for trap on ST64BV instruction

CheckST64BVEnabled()
    boolean trap = FALSE;
    bits(25) iss = Zeros(25);
    bits(2) target_el;

    if PSTATE.EL == EL0 then
        if !IsInHost() then
            trap = SCTLR_EL1.EnASR == '0';
            target_el = if EL2Enabled() && HCR_EL2.TGE == '1' then EL2 else EL1;
        else
            trap = SCTLR_EL2.EnASR == '0';
            target_el = EL2;

    if (!trap && EL2Enabled() &&
        ((PSTATE.EL == EL0 && !IsInHost()) || PSTATE.EL == EL1)) then
        trap = !IsHCRXEL2Enabled() || HCRX_EL2.EnASR == '0';
        target_el = EL2;

    if trap then LDST64BTrap(target_el, iss);
```

Library pseudocode for aarch64/exceptions/traps/LDST64BTrap

```
// LDST64BTrap()
// =============
// Trapped access to LD64B, ST64B, ST64BV and ST64BV0 instructions

LDST64BTrap(bits(2) target_el, bits(25) iss)
    bits(64) preferred_exception_return = ThisInstrAddr(64);
    vect_offset = 0x0;

    except = ExceptionSyndrome(Exception_LDST64BTrap);
    except.syndrome = iss;
    AArch64.TakeException(target_el, except, preferred_exception_return, vect_offset);

    return;
```

Library pseudocode for aarch64/exceptions/traps/WFETrapDelay

```
// WFETrapDelay()
// ==============
// Returns TRUE when delay in trap to WFE is enabled with value to amount of delay,
// FALSE otherwise.

(boolean, integer) WFETrapDelay(bits(2) target_el)
    boolean delay_enabled;
    integer delay;
    case target_el of
        when EL1
            if !IsInHost() then
                delay_enabled = SCTLR_EL1.TWEDEn == '1';
                delay         = 1 << (UInt(SCTLR_EL1.TWEDEL) + 8);
            else
                delay_enabled = SCTLR_EL2.TWEDEn == '1';
                delay         = 1 << (UInt(SCTLR_EL2.TWEDEL) + 8);
        when EL2
            assert EL2Enabled();
            delay_enabled = HCR_EL2.TWEDEn == '1';
            delay         = 1 << (UInt(HCR_EL2.TWEDEL) + 8);
        when EL3
            delay_enabled = SCR_EL3.TWEDEn == '1';
            delay         = 1 << (UInt(SCR_EL3.TWEDEL) + 8);

    return (delay_enabled, delay);
```

Library pseudocode for aarch64/exceptions/traps/WaitForEventUntilDelay

```
// WaitForEventUntilDelay()
// ========================
// Returns TRUE if WaitForEvent() returns before WFE trap delay expires,
// FALSE otherwise.

boolean WaitForEventUntilDelay(boolean delay_enabled, integer delay);
```

Library pseudocode for aarch64/functions/aborts/AArch64.FaultSyndrome

```
// AArch64.FaultSyndrome()
// =======================
// Creates an exception syndrome value for Abort and Watchpoint exceptions taken to
// an Exception level using AArch64.

(bits(25), bits(24)) AArch64.FaultSyndrome(boolean d_side, FaultRecord fault, boolean pavalid)
    assert fault.statuscode != Fault_None;

    bits(25) iss  = Zeros(25);
    bits(24) iss2 = Zeros(24);

    if HaveRASExt() && fault.statuscode == Fault_SyncExternal then
        ErrorState errstate = AArch64.PEErrorState(fault);
        iss<12:11> = AArch64.EncodeSyncErrorSyndrome(errstate);  // SET

    if d_side then
        if fault.access.acctype == AccessType_GCS then
            iss2<8> = '1';
        if HaveFeatLS64() && fault.access.ls64 then
            if (fault.statuscode IN {Fault_AccessFlag, Fault_Translation, Fault_Permission}) then
                (iss2, iss<24:14>) = LS64InstructionSyndrome();
        elsif (IsSecondStage(fault) && !fault.s2fs1walk &&
              (!IsExternalSyncAbort(fault) ||
              (!HaveRASExt() && fault.access.acctype == AccessType_TTW &&
              boolean IMPLEMENTATION_DEFINED "ISV on second stage translation table walk"))) then
            iss<24:14> = LSInstructionSyndrome();

        if HaveNV2Ext() && fault.access.acctype == AccessType_NV2 then
            iss<13> = '1';   // Fault is generated by use of VNCR_EL2

        if HaveFeatLS64() && fault.statuscode IN {Fault_AccessFlag, Fault_Translation,
                                                  Fault_Permission} then
            iss<12:11> = GetLoadStoreType();

        if fault.access.acctype IN {AccessType_DC, AccessType_IC, AccessType_AT} then
            iss<8> = '1';

        if fault.access.acctype IN {AccessType_DC, AccessType_IC, AccessType_AT} then
            iss<6> = '1';
        elsif fault.statuscode IN {Fault_HWUpdateAccessFlag, Fault_Exclusive} then
            iss<6> = bit UNKNOWN;
        elsif fault.access.atomicop && IsExternalAbort(fault) then
            iss<6> = bit UNKNOWN;
        else
            iss<6> = if fault.write then '1' else '0';

        if fault.statuscode == Fault_Permission then
            iss2<5> = if fault.dirtybit then '1' else '0';
            iss2<6> = if fault.overlay then '1' else '0';
            if iss<24> == '0' then
                iss<21> = if fault.toplevel then '1' else '0';
            iss2<7> = if fault.assuredonly then '1' else '0';
            iss2<9> = if fault.tagaccess then '1' else '0';
            iss2<10> = if fault.s1tagnotdata then '1' else '0';

    else
        if fault.access.acctype == AccessType_IFETCH && fault.statuscode == Fault_Permission then
            iss2<5> = if fault.dirtybit then '1' else '0';
            iss<21> = if fault.toplevel then '1' else '0';
            iss2<7> = if fault.assuredonly then '1' else '0';
            iss2<6> = if fault.overlay then '1' else '0';
    if IsExternalAbort(fault) then iss<9> = fault.extflag;
    iss<7> = if fault.s2fs1walk then '1' else '0';
    iss<5:0> = EncodeLDFSC(fault.statuscode, fault.level);

    return (iss, iss2);
```

Library pseudocode for aarch64/functions/aborts/EncodeGPCSC

```
// EncodeGPCSC()
// =============
// Function that gives the GPCSC code for types of GPT Fault

bits(6) EncodeGPCSC(GPCFRecord gpcf)
    assert gpcf.level IN {0,1};

    case gpcf.gpf of
        when GPCF_AddressSize return '00000':gpcf.level<0>;
        when GPCF_Walk        return '00010':gpcf.level<0>;
        when GPCF_Fail        return '00110':gpcf.level<0>;
        when GPCF_EABT        return '01010':gpcf.level<0>;
```

Library pseudocode for aarch64/functions/aborts/LS64InstructionSyndrome

```
// LS64InstructionSyndrome()
// =========================
// Returns the syndrome information and LST for a Data Abort by a
// ST64B, ST64BV, ST64BV0, or LD64B instruction. The syndrome information
// includes the ISS2, extended syndrome field.

(bits(24), bits(11)) LS64InstructionSyndrome();
```

Library pseudocode for aarch64/functions/at/AArch64.AT

```
// AArch64.AT()
// ============
// Perform address translation as per AT instructions.

AArch64.AT(bits(64) address, TranslationStage stage_in, bits(2) el_in, ATAccess ataccess)
    TranslationStage stage = stage_in;
    bits(2) el = el_in;
    bits(2) effective_nse_ns = EffectiveSCR_EL3_NSE() : EffectiveSCR_EL3_NS();
    if HaveRME() && PSTATE.EL == EL3 && effective_nse_ns == '10' && el != EL3 then
        UNDEFINED;
    // For stage 1 translation, when HCR_EL2.{E2H, TGE} is {1,1} and requested EL is EL1,
    // the EL2&0 translation regime is used.
    if HCR_EL2.<E2H, TGE> == '11' && el == EL1 && stage == TranslationStage_1 then
        el = EL2;
    if HaveEL(EL3) && stage == TranslationStage_12 && !EL2Enabled() then
        stage = TranslationStage_1;

    boolean write = ataccess IN {ATAccess_WritePAN, ATAccess_Write};
    SecurityState ss = SecurityStateAtEL(el);
    boolean pan = ataccess IN {ATAccess_ReadPAN, ATAccess_WritePAN};
    accdesc = CreateAccDescAT(ss, el, write, pan);
    aligned = TRUE;

    FaultRecord fault = NoFault(accdesc);
    Regime regime;
    if stage == TranslationStage_12 then
        regime = Regime_EL10;
    else
        regime = TranslationRegime(el);

    AddressDescriptor addrdesc;
    if (el == EL0 && ELUsingAArch32(EL1)) || (el != EL0 && ELUsingAArch32(el)) then
        if regime == Regime_EL2 || TTBCR.EAE == '1' then
            (fault, addrdesc) = AArch32.S1TranslateLD(fault, regime, address<31:0>, aligned,
                                                      accdesc);
        else
            (fault, addrdesc, -) = AArch32.S1TranslateSD(fault, regime, address<31:0>, aligned,
                                                         accdesc);
    else
        (fault, addrdesc) = AArch64.S1Translate(fault, regime, address, aligned, accdesc);

    if stage == TranslationStage_12 && fault.statuscode == Fault_None then
        boolean s1aarch64;
        if ELUsingAArch32(EL1) && regime == Regime_EL10 && EL2Enabled() then
            addrdesc.vaddress = ZeroExtend(address, 64);
            (fault, addrdesc) = AArch32.S2Translate(fault, addrdesc, aligned, accdesc);
        elsif regime == Regime_EL10 && EL2Enabled() then
            s1aarch64 = TRUE;
            (fault, addrdesc) = AArch64.S2Translate(fault, addrdesc, s1aarch64, aligned, accdesc);

    is_ATS1Ex = stage != TranslationStage_12;
    if fault.statuscode != Fault_None then
        addrdesc = CreateFaultyAddressDescriptor(address, fault);
        // Take an exception on:
        // * A Synchronous External abort occurs on translation table walk
        // * A stage 2 fault occurs on a stage 1 walk
        // * A GPC Exception (FEAT_RME)
        // * A GPF from ATS1E{1,0}* when executed from EL1 and HCR_EL2.GPF == '1' (FEAT_RME)
        if (IsExternalAbort(fault) ||
                (PSTATE.EL == EL1 && fault.s2fs1walk) ||
                (HaveRME() && fault.gpcf.gpf != GPCF_None && (
                    ReportAsGPCException(fault) ||
                    (HCR_EL2.GPF == '1' && PSTATE.EL == EL1 && el IN {EL1, EL0} && is_ATS1Ex)
                ))) then
            PAR_EL1 = bits(128) UNKNOWN;
            AArch64.Abort(address, addrdesc.fault);

    AArch64.EncodePAR(regime, is_ATS1Ex, addrdesc);
    return;
```

Library pseudocode for aarch64/functions/at/AArch64.EncodePAR

```
// AArch64.EncodePAR()
// ===================
// Encode PAR register with result of translation.

AArch64.EncodePAR(Regime regime, boolean is_ATS1Ex, AddressDescriptor addrdesc)
    PAR_EL1 = Zeros(128);
    paspace = addrdesc.paddress.paspace;

    if AArch64.isPARFormatD128(regime, is_ATS1Ex) then
        PAR_EL1.D128 = '1';
    else
        PAR_EL1.D128 = '0';

    if !IsFault(addrdesc) then
        PAR_EL1.F = '0';
        if HaveRME() then
            if regime == Regime_EL3 then
                case paspace of
                    when PAS_Secure     PAR_EL1.<NSE,NS> = '00';
                    when PAS_NonSecure  PAR_EL1.<NSE,NS> = '01';
                    when PAS_Root       PAR_EL1.<NSE,NS> = '10';
                    when PAS_Realm      PAR_EL1.<NSE,NS> = '11';

            elsif SecurityStateForRegime(regime) == SS_Secure  then
                PAR_EL1.NSE = bit UNKNOWN;
                PAR_EL1.NS = if paspace == PAS_Secure then '0' else '1';

            elsif SecurityStateForRegime(regime) == SS_Realm then
                if regime == Regime_EL10 && is_ATS1Ex then
                    PAR_EL1.NSE = bit UNKNOWN;
                    PAR_EL1.NS  = bit UNKNOWN;
                else
                    PAR_EL1.NSE = bit UNKNOWN;
                    PAR_EL1.NS  = if paspace == PAS_Realm then '0' else '1';

            else
                PAR_EL1.NSE = bit UNKNOWN;
                PAR_EL1.NS  = bit UNKNOWN;
        else
            PAR_EL1<11> = '1'; // RES1
            if SecurityStateForRegime(regime) == SS_Secure then
                PAR_EL1.NS = if paspace == PAS_Secure then '0' else '1';
            else
                PAR_EL1.NS = bit UNKNOWN;
        PAR_EL1.SH   = ReportedPARShareability(PAREncodeShareability(addrdesc.memattrs));
        if PAR_EL1.D128 == '1' then
            PAR_EL1<119:76> = addrdesc.paddress.address<55:12>;
        else
            PAR_EL1<55:12> = addrdesc.paddress.address<55:12>;
        PAR_EL1.ATTR = ReportedPARAttrs(EncodePARAttrs(addrdesc.memattrs));
        PAR_EL1<10> = bit IMPLEMENTATION_DEFINED "Non-Faulting PAR";
    else
        PAR_EL1.F   = '1';
        PAR_EL1.DirtyBit    = if addrdesc.fault.dirtybit then '1' else '0';
        PAR_EL1.Overlay     = if addrdesc.fault.overlay then '1' else '0';
        PAR_EL1.TopLevel    = if addrdesc.fault.toplevel then '1' else '0';
        PAR_EL1.AssuredOnly = if addrdesc.fault.assuredonly then '1' else '0';
        PAR_EL1.FST = AArch64.PARFaultStatus(addrdesc.fault);
        PAR_EL1.PTW = if addrdesc.fault.s2fs1walk then '1' else '0';
        PAR_EL1.S   = if addrdesc.fault.secondstage then '1' else '0';
        PAR_EL1<11> = '1'; // RES1
        PAR_EL1<63:48> = bits(16) IMPLEMENTATION_DEFINED "Faulting PAR";
    return;
```

Library pseudocode for aarch64/functions/at/AArch64.PARFaultStatus

```
// AArch64.PARFaultStatus()
// ========================
// Fault status field decoding of 64-bit PAR.

bits(6) AArch64.PARFaultStatus(FaultRecord fault)
    bits(6) fst;

    if fault.statuscode == Fault_Domain then
        // Report Domain fault
        assert fault.level IN {1,2};
        fst<1:0> = if fault.level == 1 then '01' else '10';
        fst<5:2> = '1111';
    else
        fst = EncodeLDFSC(fault.statuscode, fault.level);
    return fst;
```

Library pseudocode for aarch64/functions/at/AArch64.isPARFormatD128

```
// AArch64.isPARFormatD128()
// =========================
// Check if last stage of translation uses VMSAv9-128.
// Last stage of translation is stage 2 if enabled, else it is stage 1.

boolean AArch64.isPARFormatD128(Regime regime, boolean is_ATS1Ex)
    boolean isPARFormatD128;
    // Regime_EL2 does not support VMSAv9-128
    if regime == Regime_EL2 || !Have128BitDescriptorExt() then
        isPARFormatD128 = FALSE;
    else
        isPARFormatD128 = FALSE;
        case regime of
            when Regime_EL3
                isPARFormatD128 = TCR_EL3.D128 == '1';
            when Regime_EL20
                isPARFormatD128 = TCR2_EL2.D128 == '1';
            when Regime_EL10
                if is_ATS1Ex || !EL2Enabled() || HCR_EL2.<VM,DC> == '00' then
                    isPARFormatD128 = TCR2_EL1.D128 == '1';
                else
                    isPARFormatD128 = VTCR_EL2.D128 == '1';

    return isPARFormatD128;
```

Library pseudocode for aarch64/functions/at/GetPAR_EL1_D128

```
// GetPAR_EL1_D128()
// =================
// Query the PAR_EL1.D128 field

bit GetPAR_EL1_D128()
    bit D128;

    D128 = PAR_EL1.D128;
    return D128;
```

Library pseudocode for aarch64/functions/at/GetPAR_EL1_F

```
// GetPAR_EL1_F()
// ==============
// Query the PAR_EL1.F field.

bit GetPAR_EL1_F()
    bit F;

    F = PAR_EL1.F;
    return F;
```

Library pseudocode for aarch64/functions/barrierop/MemBarrierOp

```
// MemBarrierOp
// ============
// Memory barrier instruction types.

enumeration MemBarrierOp   {  MemBarrierOp_DSB         // Data Synchronization Barrier
                            , MemBarrierOp_DMB         // Data Memory Barrier
                            , MemBarrierOp_ISB         // Instruction Synchronization Barrier
                            , MemBarrierOp_SSBB        // Speculative Synchronization Barrier to VA
                            , MemBarrierOp_PSSBB       // Speculative Synchronization Barrier to PA
                            , MemBarrierOp_SB          // Speculation Barrier
                           };
```

Library pseudocode for aarch64/functions/bfxpreferred/BFXPreferred

```
// BFXPreferred()
// ==============
//
// Return TRUE if UBFX or SBFX is the preferred disassembly of a
// UBFM or SBFM bitfield instruction. Must exclude more specific
// aliases UBFIZ, SBFIZ, UXT[BH], SXT[BHW], LSL, LSR and ASR.

boolean BFXPreferred(bit sf, bit uns, bits(6) imms, bits(6) immr)

    // must not match UBFIZ/SBFIX alias
    if UInt(imms) < UInt(immr) then
        return FALSE;

    // must not match LSR/ASR/LSL alias (imms == 31 or 63)
    if imms == sf:'11111' then
        return FALSE;

    // must not match UXTx/SXTx alias
    if immr == '000000' then
        // must not match 32-bit UXT[BH] or SXT[BH]
        if sf == '0' && imms IN {'000111', '001111'} then
            return FALSE;
        // must not match 64-bit SXT[BHW]
        if sf:uns == '10' && imms IN {'000111', '001111', '011111'} then
            return FALSE;

    // must be UBFX/SBFX alias
    return TRUE;
```

Library pseudocode for aarch64/functions/bitmasks/AltDecodeBitMasks

```
// AltDecodeBitMasks()
// ===================
// Alternative but logically equivalent implementation of DecodeBitMasks() that
// uses simpler primitives to compute tmask and wmask.

(bits(M), bits(M)) AltDecodeBitMasks(bit immN, bits(6) imms, bits(6) immr,
                                     boolean immediate, integer M)
    bits(64) tmask, wmask;
    bits(6) tmask_and, wmask_and;
    bits(6) tmask_or, wmask_or;
    bits(6) levels;

    // Compute log2 of element size
    // 2^len must be in range [2, M]
    len = HighestSetBit(immN:NOT(imms));
    if len < 1 then UNDEFINED;
    assert M >= (1 << len);

    // Determine s, r and s - r parameters
    levels = ZeroExtend(Ones(len), 6);

    // For logical immediates an all-ones value of s is reserved
    // since it would generate a useless all-ones result (many times)
    if immediate && (imms AND levels) == levels then
        UNDEFINED;

    s = UInt(imms AND levels);
    r = UInt(immr AND levels);
    diff = s - r;    // 6-bit subtract with borrow

    // Compute "top mask"
    tmask_and = diff<5:0> OR NOT(levels);
    tmask_or  = diff<5:0> AND levels;

    tmask = Ones(64);
    tmask = ((tmask
              AND Replicate(Replicate(tmask_and<0>, 1) : Ones(1), 32))
               OR  Replicate(Zeros(1) : Replicate(tmask_or<0>, 1), 32));
    // optimization of first step:
    // tmask = Replicate(tmask_and<0> : '1', 32);
    tmask = ((tmask
              AND Replicate(Replicate(tmask_and<1>, 2) : Ones(2), 16))
               OR  Replicate(Zeros(2) : Replicate(tmask_or<1>, 2), 16));
    tmask = ((tmask
              AND Replicate(Replicate(tmask_and<2>, 4) : Ones(4), 8))
               OR  Replicate(Zeros(4) : Replicate(tmask_or<2>, 4), 8));
    tmask = ((tmask
              AND Replicate(Replicate(tmask_and<3>, 8) : Ones(8), 4))
               OR  Replicate(Zeros(8) : Replicate(tmask_or<3>, 8), 4));
    tmask = ((tmask
              AND Replicate(Replicate(tmask_and<4>, 16) : Ones(16), 2))
               OR  Replicate(Zeros(16) : Replicate(tmask_or<4>, 16), 2));
    tmask = ((tmask
              AND Replicate(Replicate(tmask_and<5>, 32) : Ones(32), 1))
               OR  Replicate(Zeros(32) : Replicate(tmask_or<5>, 32), 1));

    // Compute "wraparound mask"
    wmask_and = immr OR NOT(levels);
    wmask_or  = immr AND levels;

    wmask = Zeros(64);
    wmask = ((wmask
              AND Replicate(Ones(1) : Replicate(wmask_and<0>, 1), 32))
               OR  Replicate(Replicate(wmask_or<0>, 1) : Zeros(1), 32));
    // optimization of first step:
    // wmask = Replicate(wmask_or<0> : '0', 32);
    wmask = ((wmask
              AND Replicate(Ones(2) : Replicate(wmask_and<1>, 2), 16))
               OR  Replicate(Replicate(wmask_or<1>, 2) : Zeros(2), 16));
    wmask = ((wmask
```

```
              AND Replicate(Ones(4) : Replicate(wmask_and<2>, 4), 8))
               OR  Replicate(Replicate(wmask_or<2>, 4) : Zeros(4), 8));
    wmask = ((wmask
              AND Replicate(Ones(8) : Replicate(wmask_and<3>, 8), 4))
               OR  Replicate(Replicate(wmask_or<3>, 8) : Zeros(8), 4));
    wmask = ((wmask
              AND Replicate(Ones(16) : Replicate(wmask_and<4>, 16), 2))
               OR  Replicate(Replicate(wmask_or<4>, 16) : Zeros(16), 2));
    wmask = ((wmask
              AND Replicate(Ones(32) : Replicate(wmask_and<5>, 32), 1))
               OR  Replicate(Replicate(wmask_or<5>, 32) : Zeros(32), 1));

    if diff<6> != '0' then // borrow from s - r
        wmask = wmask AND tmask;
    else
        wmask = wmask OR tmask;

    return (wmask<M-1:0>, tmask<M-1:0>);
```

Library pseudocode for aarch64/functions/bitmasks/DecodeBitMasks

```
// DecodeBitMasks()
// ================
// Decode AArch64 bitfield and logical immediate masks which use a similar encoding structure

(bits(M), bits(M)) DecodeBitMasks(bit immN, bits(6) imms, bits(6) immr,
                                  boolean immediate, integer M)
    bits(M) tmask, wmask;
    bits(6) levels;

    // Compute log2 of element size
    // 2^len must be in range [2, M]
    len = HighestSetBit(immN:NOT(imms));
    if len < 1 then UNDEFINED;
    assert M >= (1 << len);

    // Determine s, r and s - r parameters
    levels = ZeroExtend(Ones(len), 6);

    // For logical immediates an all-ones value of s is reserved
    // since it would generate a useless all-ones result (many times)
    if immediate && (imms AND levels) == levels then
        UNDEFINED;

    s = UInt(imms AND levels);
    r = UInt(immr AND levels);
    diff = s - r;    // 6-bit subtract with borrow

    esize = 1 << len;
    d = UInt(diff<len-1:0>);
    welem = ZeroExtend(Ones(s + 1), esize);
    telem = ZeroExtend(Ones(d + 1), esize);
    wmask = Replicate(ROR(welem, r), M DIV esize);
    tmask = Replicate(telem, M DIV esize);
    return (wmask, tmask);
```

Library pseudocode for aarch64/functions/cache/AArch64.DataMemZero

```
// AArch64.DataMemZero()
// =====================
// Write Zero to data memory.

AArch64.DataMemZero(bits(64) regval, bits(64) vaddress, AccessDescriptor accdesc_in, integer size)
    AccessDescriptor accdesc = accdesc_in;

    // If the instruction targets tags as a payload, confer with system register configuration
    // which may override this.
    if HaveMTE2Ext() && accdesc.tagaccess then
        accdesc.tagaccess = AArch64.AllocationTagAccessIsEnabled(accdesc.el);

    // If the instruction encoding permits tag checking, confer with system register configuration
    // which may override this.
    if HaveMTE2Ext() && accdesc.tagchecked then
        accdesc.tagchecked = AArch64.AccessIsTagChecked(vaddress, accdesc);

    boolean aligned = TRUE;
    AddressDescriptor memaddrdesc = AArch64.TranslateAddress(vaddress, accdesc, aligned, size);

    if IsFault(memaddrdesc) then
        if IsDebugException(memaddrdesc.fault) then
            AArch64.Abort(vaddress, memaddrdesc.fault);
        else
            AArch64.Abort(regval, memaddrdesc.fault);

    if HaveTME() then
        if accdesc.transactional && !MemHasTransactionalAccess(memaddrdesc.memattrs) then
            FailTransaction(TMFailure_IMP, FALSE);

    for i = 0 to size-1
        if HaveMTE2Ext() && accdesc.tagchecked then
            bits(4) ptag = AArch64.PhysicalTag(vaddress);
            if !AArch64.CheckTag(memaddrdesc, accdesc, ptag) then
                if (boolean IMPLEMENTATION_DEFINED
                      "DC_ZVA tag fault reported with lowest faulting address") then
                    AArch64.TagCheckFault(vaddress, accdesc);
                else
                    AArch64.TagCheckFault(regval, accdesc);
        memstatus = PhysMemWrite(memaddrdesc, 1, accdesc, Zeros(8));
        if IsFault(memstatus) then
            HandleExternalWriteAbort(memstatus, memaddrdesc, 1, accdesc);

        memaddrdesc.paddress.address = memaddrdesc.paddress.address + 1;
    return;
```

Library pseudocode for aarch64/functions/cache/AArch64.TagMemZero

```
// AArch64.TagMemZero()
// ====================
// Write Zero to tag memory.

AArch64.TagMemZero(bits(64) regval, bits(64) vaddress, AccessDescriptor accdesc_in, integer size)
    assert accdesc_in.tagaccess && !accdesc_in.tagchecked;

    AccessDescriptor accdesc = accdesc_in;

    integer count = size >> LOG2_TAG_GRANULE;
    bits(4) tag = AArch64.AllocationTagFromAddress(vaddress);
    boolean aligned = IsAligned(vaddress, TAG_GRANULE);

    // Stores of allocation tags must be aligned
    if !aligned then
        AArch64.Abort(vaddress, AlignmentFault(accdesc));

    if HaveMTE2Ext() then
        accdesc.tagaccess = AArch64.AllocationTagAccessIsEnabled(accdesc.el);

    memaddrdesc = AArch64.TranslateAddress(vaddress, accdesc, aligned, size);

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        if IsDebugException(memaddrdesc.fault) then
            AArch64.Abort(vaddress, memaddrdesc.fault);
        else
            AArch64.Abort(regval, memaddrdesc.fault);

    if !accdesc.tagaccess || memaddrdesc.memattrs.tags != MemTag_AllocationTagged then
        return;

    for i = 0 to count-1
        memstatus = PhysMemTagWrite(memaddrdesc, accdesc, tag);
        if IsFault(memstatus) then
            HandleExternalWriteAbort(memstatus, memaddrdesc, 1, accdesc);

        memaddrdesc.paddress.address = memaddrdesc.paddress.address + TAG_GRANULE;

    return;
```

Library pseudocode for aarch64/functions/compareop/CompareOp

```
// CompareOp
// =========
// Vector compare instruction types.

enumeration CompareOp   {CompareOp_GT, CompareOp_GE, CompareOp_EQ,
                         CompareOp_LE, CompareOp_LT};
```

Library pseudocode for aarch64/functions/countop/CountOp

```
// CountOp
// =======
// Bit counting instruction types.

enumeration CountOp     {CountOp_CLZ, CountOp_CLS, CountOp_CNT};
```

Library pseudocode for aarch64/functions/d128/IsD128Enabled

```
// IsD128Enabled()
// ===============
// Returns true if 128-bit page descriptor is enabled

boolean IsD128Enabled(bits(2) el)
    boolean d128enabled;
    if Have128BitDescriptorExt() then
        case el of
            when EL0
                if !ELIsInHost(EL0) then
                    d128enabled = IsTCR2EL1Enabled() && TCR2_EL1.D128 == '1';
                else
                    d128enabled = IsTCR2EL2Enabled() && TCR2_EL2.D128 == '1';
            when EL1
                d128enabled = IsTCR2EL1Enabled() && TCR2_EL1.D128 == '1';
            when EL2
                d128enabled = IsTCR2EL2Enabled() && HCR_EL2.E2H == '1' && TCR2_EL2.D128 == '1';
            when EL3
                d128enabled = TCR_EL3.D128 == '1';
    else
        d128enabled = FALSE;

    return d128enabled;
```

Library pseudocode for aarch64/functions/dc/AArch64.DC

```
// AArch64.DC()
// ============
// Perform Data Cache Operation.

AArch64.DC(bits(64) regval, CacheType cachetype, CacheOp cacheop, CacheOpScope opscope_in)
    CacheOpScope opscope = opscope_in;
    CacheRecord cache;

    cache.acctype   = AccessType_DC;
    cache.cachetype = cachetype;
    cache.cacheop   = cacheop;
    cache.opscope   = opscope;

    if opscope == CacheOpScope_SetWay then
        ss = SecurityStateAtEL(PSTATE.EL);
        cache.cpas = CPASAtSecurityState(ss);
        cache.shareability = Shareability_NSH;
        (cache.setnum, cache.waynum, cache.level) = DecodeSW(regval, cachetype);
        if (cacheop == CacheOp_Invalidate && PSTATE.EL == EL1 && EL2Enabled() &&
          (HCR_EL2.SWIO == '1' || HCR_EL2.<DC,VM> != '00')) then
            cache.cacheop = CacheOp_CleanInvalidate;

        CACHE_OP(cache);
        return;

    if EL2Enabled() && !IsInHost() then
        if PSTATE.EL IN {EL0, EL1} then
            cache.is_vmid_valid = TRUE;
            cache.vmid          = VMID[];
        else
            cache.is_vmid_valid = FALSE;
    else
        cache.is_vmid_valid = FALSE;

    if PSTATE.EL == EL0 then
        cache.is_asid_valid = TRUE;
        cache.asid          = ASID[];
    else
        cache.is_asid_valid = FALSE;

    if (opscope == CacheOpScope_PoDP &&
          boolean IMPLEMENTATION_DEFINED "Memory system does not supports PoDP") then
        opscope = CacheOpScope_PoP;
    if (opscope == CacheOpScope_PoP &&
          boolean IMPLEMENTATION_DEFINED "Memory system does not supports PoP") then
        opscope = CacheOpScope_PoC;
    vaddress = regval;

    size = 0;        // by default no watchpoint address
    if cacheop == CacheOp_Invalidate then
        size = integer IMPLEMENTATION_DEFINED "Data Cache Invalidate Watchpoint Size";
        assert size >= 4*(2^(UInt(CTR_EL0.DminLine))) && size <= 2048;
        assert UInt(size<32:0> AND (size-1)<32:0>) == 0; // size is power of 2
        vaddress = Align(regval, size);

    if DCInstNeedsTranslation(opscope) then
        cache.vaddress  = vaddress;
        boolean aligned = TRUE;
        AccessDescriptor accdesc = CreateAccDescDC(cache);
        AddressDescriptor memaddrdesc = AArch64.TranslateAddress(vaddress, accdesc, aligned, size);
        if IsFault(memaddrdesc) then
            AArch64.Abort(regval, memaddrdesc.fault);

        cache.translated = TRUE;
        cache.paddress   = memaddrdesc.paddress;
        cache.cpas       = CPASAtPAS(memaddrdesc.paddress.paspace);
        if opscope IN {CacheOpScope_PoC, CacheOpScope_PoP, CacheOpScope_PoDP} then
            cache.shareability = memaddrdesc.memattrs.shareability;
        else
            cache.shareability = Shareability_NSH;
```

```
    elsif opscope == CacheOpScope_PoE then
        cache.translated       = TRUE;
        cache.shareability     = Shareability_OSH;
        cache.paddress.address = regval<55:0>;
        cache.paddress.paspace = DecodePASpace(regval<62>, regval<63>);
        cache.cpas             = CPASAtPAS(cache.paddress.paspace);

        // If a Reserved encoding is selected, the instruction is permitted to be treated as a NOP.
        if cache.paddress.paspace != PAS_Realm then
            EndOfInstruction();

        if boolean IMPLEMENTATION_DEFINED "Apply granule protection check on DC to PoE" then
            AddressDescriptor memaddrdesc;
            AccessDescriptor accdesc = CreateAccDescDC(cache);
            memaddrdesc.paddress     = cache.paddress;
            memaddrdesc.fault.gpcf   = GranuleProtectionCheck(memaddrdesc, accdesc);

            if memaddrdesc.fault.gpcf.gpf != GPCF_None then
                memaddrdesc.fault.statuscode = Fault_GPCFOnOutput;
                memaddrdesc.fault.paddress   = memaddrdesc.paddress;
                AArch64.Abort(bits(64) UNKNOWN, memaddrdesc.fault);
    elsif opscope == CacheOpScope_PoPA then
        cache.translated   = TRUE;
        cache.shareability = Shareability_OSH;
        cache.paddress.address = regval<55:0>;
        cache.paddress.paspace = DecodePASpace(regval<62>, regval<63>);
        cache.cpas = CPASAtPAS(cache.paddress.paspace);
    else
        cache.vaddress     = vaddress;
        cache.translated   = FALSE;
        cache.shareability = Shareability UNKNOWN;
        cache.paddress     = FullAddress UNKNOWN;

    if (cacheop == CacheOp_Invalidate && PSTATE.EL == EL1 && EL2Enabled() &&
            HCR_EL2.<DC,VM> != '00') then
        cache.cacheop = CacheOp_CleanInvalidate;

    // If Secure state is not implemented, but RME is, the instruction acts as a NOP
    if cache.translated && cache.cpas == CPAS_Secure && !HaveSecureState() then
        return;

    CACHE_OP(cache);
    return;
```

Library pseudocode for aarch64/functions/dc/AArch64.MemZero

```
// AArch64.MemZero()
// =================

AArch64.MemZero(bits(64) regval, CacheType cachetype)
    integer size = 4*(2^(UInt(DCZID_EL0.BS)));
    assert size <= MAX_ZERO_BLOCK_SIZE;
    if HaveMTE2Ext() then
        assert size >= TAG_GRANULE;

    bits(64) vaddress = Align(regval, size);

    boolean tagaccess = cachetype IN {CacheType_Tag, CacheType_Data_Tag};
    boolean tagchecked = cachetype == CacheType_Data;
    AccessDescriptor accdesc = CreateAccDescDCZero(tagaccess, tagchecked);

    if cachetype IN {CacheType_Tag, CacheType_Data_Tag} then
        AArch64.TagMemZero(regval, vaddress, accdesc, size);

    if cachetype IN {CacheType_Data, CacheType_Data_Tag} then
        AArch64.DataMemZero(regval, vaddress, accdesc, size);
    return;
```

Library pseudocode for aarch64/functions/dc/MemZero

```
constant integer MAX_ZERO_BLOCK_SIZE = 2048;
```

Library pseudocode for aarch64/functions/eret/AArch64.ExceptionReturn

```
// AArch64.ExceptionReturn()
// =========================

AArch64.ExceptionReturn(bits(64) new_pc_in, bits(64) spsr)
    bits(64) new_pc = new_pc_in;
    if HaveTME() && TSTATE.depth > 0 then
        FailTransaction(TMFailure_ERR, FALSE);

    if HaveIESB() then
        sync_errors = SCTLR[].IESB == '1';
        if HaveDoubleFaultExt() then
            sync_errors = sync_errors || (SCR_EL3.<EA,NMEA> == '11' && PSTATE.EL == EL3);
        if sync_errors then
            SynchronizeErrors();
            iesb_req = TRUE;
            TakeUnmaskedPhysicalSErrorInterrupts(iesb_req);
    SynchronizeContext();

    // Attempts to change to an illegal state will invoke the Illegal Execution state mechanism
    bits(2) source_el = PSTATE.EL;
    boolean illegal_psr_state = IllegalExceptionReturn(spsr);
    SetPSTATEFromPSR(spsr, illegal_psr_state);
    ClearExclusiveLocal(ProcessorID());
    SendEventLocal();

    if illegal_psr_state && spsr<4> == '1' then
        // If the exception return is illegal, PC[63:32,1:0] are UNKNOWN
        new_pc<63:32> = bits(32) UNKNOWN;
        new_pc<1:0> = bits(2) UNKNOWN;
    elsif UsingAArch32() then                // Return to AArch32
        // ELR_ELx[1:0] or ELR_ELx[0] are treated as being 0, depending on the
        // target instruction set state
        if PSTATE.T == '1' then
            new_pc<0> = '0';                 // T32
        else
            new_pc<1:0> = '00';              // A32
    else                                     // Return to AArch64
        // ELR_ELx[63:56] might include a tag
        new_pc = AArch64.BranchAddr(new_pc, PSTATE.EL);

    if HaveBRBExt() then
        BRBEExceptionReturn(new_pc, source_el);

    if UsingAArch32() then
        if HaveSME() && PSTATE.SM == '1' then ResetSVEState();

        // 32 most significant bits are ignored.
        boolean branch_conditional = FALSE;
        BranchTo(new_pc<31:0>, BranchType_ERET, branch_conditional);
    else
        BranchToAddr(new_pc, BranchType_ERET);

    CheckExceptionCatch(FALSE);              // Check for debug event on exception return
```

Library pseudocode for aarch64/functions/exclusive/AArch64.ExclusiveMonitorsPass

```
// AArch64.ExclusiveMonitorsPass()
// ===============================
// Return TRUE if the Exclusives monitors for the current PE include all of the addresses
// associated with the virtual address region of size bytes starting at address.
// The immediately following memory write must be to the same addresses.

boolean AArch64.ExclusiveMonitorsPass(bits(64) address, integer size)

    // It is IMPLEMENTATION DEFINED whether the detection of memory aborts happens
    // before or after the check on the local Exclusives monitor. As a result a failure
    // of the local monitor can occur on some implementations even if the memory
    // access would give an memory abort.
    boolean acqrel = FALSE;
    boolean tagchecked = FALSE;
    AccessDescriptor accdesc = CreateAccDescExLDST(MemOp_STORE, acqrel, tagchecked);
    boolean aligned = IsAligned(address, size);

    if !aligned && AArch64.UnalignedAccessFaults(accdesc, address, size) then
        AArch64.Abort(address, AlignmentFault(accdesc));

    if !AArch64.IsExclusiveVA(address, ProcessorID(), size) then
        return FALSE;

    memaddrdesc = AArch64.TranslateAddress(address, accdesc, aligned, size);

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        AArch64.Abort(address, memaddrdesc.fault);

    passed = IsExclusiveLocal(memaddrdesc.paddress, ProcessorID(), size);
    ClearExclusiveLocal(ProcessorID());

    if passed && memaddrdesc.memattrs.shareability != Shareability_NSH then
        passed = IsExclusiveGlobal(memaddrdesc.paddress, ProcessorID(), size);

    return passed;
```

Library pseudocode for aarch64/functions/exclusive/AArch64.IsExclusiveVA

```
// AArch64.IsExclusiveVA()
// =======================
// An optional IMPLEMENTATION DEFINED test for an exclusive access to a virtual
// address region of size bytes starting at address.
//
// It is permitted (but not required) for this function to return FALSE and
// cause a store exclusive to fail if the virtual address region is not
// totally included within the region recorded by MarkExclusiveVA().
//
// It is always safe to return TRUE which will check the physical address only.

boolean AArch64.IsExclusiveVA(bits(64) address, integer processorid, integer size);
```

Library pseudocode for aarch64/functions/exclusive/AArch64.MarkExclusiveVA

```
// AArch64.MarkExclusiveVA()
// =========================
// Optionally record an exclusive access to the virtual address region of size bytes
// starting at address for processorid.

AArch64.MarkExclusiveVA(bits(64) address, integer processorid, integer size);
```

Library pseudocode for aarch64/functions/exclusive/AArch64.SetExclusiveMonitors

```
// AArch64.SetExclusiveMonitors()
// ==============================
// Sets the Exclusives monitors for the current PE to record the addresses associated
// with the virtual address region of size bytes starting at address.

AArch64.SetExclusiveMonitors(bits(64) address, integer size)
    boolean acqrel = FALSE;
    boolean tagchecked = FALSE;
    AccessDescriptor accdesc = CreateAccDescExLDST(MemOp_LOAD, acqrel, tagchecked);
    boolean aligned = IsAligned(address, size);

    if !aligned && AArch64.UnalignedAccessFaults(accdesc, address, size) then
        AArch64.Abort(address, AlignmentFault(accdesc));

    memaddrdesc = AArch64.TranslateAddress(address, accdesc, aligned, size);

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        return;

    if memaddrdesc.memattrs.shareability != Shareability_NSH then
        MarkExclusiveGlobal(memaddrdesc.paddress, ProcessorID(), size);

    MarkExclusiveLocal(memaddrdesc.paddress, ProcessorID(), size);

    AArch64.MarkExclusiveVA(address, ProcessorID(), size);
```

Library pseudocode for aarch64/functions/extendreg/DecodeRegExtend

```
// DecodeRegExtend()
// =================
// Decode a register extension option

ExtendType DecodeRegExtend(bits(3) op)
    case op of
        when '000' return ExtendType_UXTB;
        when '001' return ExtendType_UXTH;
        when '010' return ExtendType_UXTW;
        when '011' return ExtendType_UXTX;
        when '100' return ExtendType_SXTB;
        when '101' return ExtendType_SXTH;
        when '110' return ExtendType_SXTW;
        when '111' return ExtendType_SXTX;
```

Library pseudocode for aarch64/functions/extendreg/ExtendReg

```
// ExtendReg()
// ===========
// Perform a register extension and shift

bits(N) ExtendReg(integer reg, ExtendType exttype, integer shift, integer N)
    assert shift >= 0 && shift <= 4;
    bits(N) val = X[reg, N];
    boolean unsigned;
    integer len;

    case exttype of
        when ExtendType_SXTB unsigned = FALSE; len = 8;
        when ExtendType_SXTH unsigned = FALSE; len = 16;
        when ExtendType_SXTW unsigned = FALSE; len = 32;
        when ExtendType_SXTX unsigned = FALSE; len = 64;
        when ExtendType_UXTB unsigned = TRUE;  len = 8;
        when ExtendType_UXTH unsigned = TRUE;  len = 16;
        when ExtendType_UXTW unsigned = TRUE;  len = 32;
        when ExtendType_UXTX unsigned = TRUE;  len = 64;

    // Note the extended width of the intermediate value and
    // that sign extension occurs from bit <len+shift-1>, not
    // from bit <len-1>. This is equivalent to the instruction
    //   [SU]BFIZ Rtmp, Rreg, #shift, #len
    // It may also be seen as a sign/zero extend followed by a shift:
    //   LSL(Extend(val<len-1:0>, N, unsigned), shift);

    len = Min(len, N - shift);
    return Extend(val<len-1:0> : Zeros(shift), N, unsigned);
```

Library pseudocode for aarch64/functions/extendreg/ExtendType

```
// ExtendType
// ==========
// AArch64 register extend and shift.

enumeration ExtendType  {ExtendType_SXTB, ExtendType_SXTH, ExtendType_SXTW, ExtendType_SXTX,
                         ExtendType_UXTB, ExtendType_UXTH, ExtendType_UXTW, ExtendType_UXTX};
```

Library pseudocode for aarch64/functions/fpconvop/FPConvOp

```
// FPConvOp
// ========
// Floating-point convert/move instruction types.

enumeration FPConvOp    {FPConvOp_CVT_FtoI, FPConvOp_CVT_ItoF,
                         FPConvOp_MOV_FtoI, FPConvOp_MOV_ItoF
                         , FPConvOp_CVT_FtoI_JS
};
```

Library pseudocode for aarch64/functions/fpmaxminop/FPMaxMinOp

```
// FPMaxMinOp
// ==========
// Floating-point min/max instruction types.

enumeration FPMaxMinOp  {FPMaxMinOp_MAX, FPMaxMinOp_MIN,
                         FPMaxMinOp_MAXNUM, FPMaxMinOp_MINNUM};
```

Library pseudocode for aarch64/functions/fpunaryop/FPUnaryOp

```
// FPUnaryOp
// =========
// Floating-point unary instruction types.

enumeration FPUnaryOp   {FPUnaryOp_ABS, FPUnaryOp_MOV,
                         FPUnaryOp_NEG, FPUnaryOp_SQRT};
```

Library pseudocode for aarch64/functions/fusedrstep/FPRSqrtStepFused

```
// FPRSqrtStepFused()
// ==================

bits(N) FPRSqrtStepFused(bits(N) op1_in, bits(N) op2)
    assert N IN {16, 32, 64};
    bits(N) result;
    bits(N) op1 = op1_in;
    boolean done;
    FPCRType fpcr = FPCR[];
    op1 = FPNeg(op1);
    boolean altfp = HaveAltFP() && fpcr.AH == '1';
    boolean fpexc = !altfp;                         // Generate no floating-point exceptions
    if altfp then fpcr.<FIZ,FZ> = '11';             // Flush denormal input and output to zero
    if altfp then fpcr.RMode = '00';                // Use RNE rounding mode

    (type1,sign1,value1) = FPUnpack(op1, fpcr, fpexc);
    (type2,sign2,value2) = FPUnpack(op2, fpcr, fpexc);
    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr, fpexc);
    FPRounding rounding = FPRoundingMode(fpcr);

    if !done then
        inf1 = (type1 == FPType_Infinity);
        inf2 = (type2 == FPType_Infinity);
        zero1 = (type1 == FPType_Zero);
        zero2 = (type2 == FPType_Zero);

        if (inf1 && zero2) || (zero1 && inf2) then
            result = FPOnePointFive('0', N);
        elsif inf1 || inf2 then
            result = FPInfinity(sign1 EOR sign2, N);
        else
            // Fully fused multiply-add and halve
            result_value = (3.0 + (value1 * value2)) / 2.0;
            if result_value == 0.0 then
                // Sign of exact zero result depends on rounding mode
                sign = if rounding == FPRounding_NEGINF then '1' else '0';
                result = FPZero(sign, N);
            else
                result = FPRound(result_value, fpcr, rounding, fpexc, N);

    return result;
```

Library pseudocode for aarch64/functions/fusedrstep/FPRecipStepFused

```
// FPRecipStepFused()
// ==================

bits(N) FPRecipStepFused(bits(N) op1_in, bits(N) op2)
    assert N IN {16, 32, 64};
    bits(N) op1 = op1_in;
    bits(N) result;
    boolean done;
    FPCRType fpcr = FPCR[];
    op1 = FPNeg(op1);

    boolean altfp = HaveAltFP() && fpcr.AH == '1';
    boolean fpexc = !altfp;                          // Generate no floating-point exceptions
    if altfp then fpcr.<FIZ,FZ> = '11';              // Flush denormal input and output to zero
    if altfp then fpcr.RMode    = '00';              // Use RNE rounding mode

    (type1,sign1,value1) = FPUnpack(op1, fpcr, fpexc);
    (type2,sign2,value2) = FPUnpack(op2, fpcr, fpexc);
    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr, fpexc);
    FPRounding rounding = FPRoundingMode(fpcr);

    if !done then
        inf1  = (type1 == FPType_Infinity);
        inf2  = (type2 == FPType_Infinity);
        zero1 = (type1 == FPType_Zero);
        zero2 = (type2 == FPType_Zero);

        if (inf1 && zero2) || (zero1 && inf2) then
            result = FPTwo('0', N);
        elsif inf1 || inf2 then
            result = FPInfinity(sign1 EOR sign2, N);
        else
            // Fully fused multiply-add
            result_value = 2.0 + (value1 * value2);
            if result_value == 0.0 then
                // Sign of exact zero result depends on rounding mode
                sign = if rounding == FPRounding_NEGINF then '1' else '0';
                result = FPZero(sign, N);
            else
                result = FPRound(result_value, fpcr, rounding, fpexc, N);

    return result;
```

Library pseudocode for aarch64/functions/gcs/AddGCSExRecord

```
// AddGCSExRecord()
// ================
// Generates and then writes an exception record to the
// current Guarded control stack.

AddGCSExRecord(bits(64) elr, bits(64) spsr, bits(64) lr)
    bits(64) ptr;
    AccessDescriptor accdesc = CreateAccDescGCS(PSTATE.EL, MemOp_STORE);

    ptr = GetCurrentGCSPointer();

    // Store the record
    Mem[ptr-8, 8, accdesc] = lr;
    Mem[ptr-16, 8, accdesc] = spsr;
    Mem[ptr-24, 8, accdesc] = elr;
    Mem[ptr-32, 8, accdesc] = Zeros(60):'1001';

    // Decrement the pointer value
    ptr = ptr - 32;

    SetCurrentGCSPointer(ptr);
    return;
```

Library pseudocode for aarch64/functions/gcs/AddGCSRecord

```
// AddGCSRecord()
// ==============
// Generates and then writes a record to the current Guarded
// control stack.

AddGCSRecord(bits(64) vaddress)
    bits(64) ptr;
    AccessDescriptor accdesc = CreateAccDescGCS(PSTATE.EL, MemOp_STORE);

    ptr = GetCurrentGCSPointer();

    // Store the record
    Mem[ptr-8, 8, accdesc] = vaddress;

    // Decrement the pointer value
    ptr = ptr - 8;

    SetCurrentGCSPointer(ptr);
    return;
```

Library pseudocode for aarch64/functions/gcs/CheckGCSExRecord

```
// CheckGCSExRecord()
// ==================
// Validates the provided values against the top entry of the
// current Guarded control stack.

CheckGCSExRecord(bits(64) elr, bits(64) spsr, bits(64) lr, GCSInstruction gcsinst_type)
    bits(64) ptr;
    AccessDescriptor accdesc = CreateAccDescGCS(PSTATE.EL, MemOp_LOAD);
    ptr = GetCurrentGCSPointer();

    // Check the lowest doubleword is correctly formatted
    bits(64) recorded_first_dword = Mem[ptr, 8, accdesc];
    if recorded_first_dword != Zeros(60):'1001' then
        GCSDataCheckException(gcsinst_type);

    // Check the ELR matches the recorded value
    bits(64) recorded_elr = Mem[ptr+8, 8, accdesc];
    if recorded_elr != elr then
        GCSDataCheckException(gcsinst_type);

    // Check the SPSR matches the recorded value
    bits(64) recorded_spsr = Mem[ptr+16, 8, accdesc];
    if recorded_spsr != spsr then
        GCSDataCheckException(gcsinst_type);

    // Check the LR matches the recorded value
    bits(64) recorded_lr = Mem[ptr+24, 8, accdesc];
    if recorded_lr != lr then
        GCSDataCheckException(gcsinst_type);

    // Increment the pointer value
    ptr = ptr + 32;

    SetCurrentGCSPointer(ptr);
    return;
```

Library pseudocode for aarch64/functions/gcs/CheckGCSSTREnabled

```
// CheckGCSSTREnabled()
// ====================
// Trap GCSSTR or GCSSTTR instruction if trapping is enabled.

CheckGCSSTREnabled()
    case PSTATE.EL of
        when EL0
            if GCSCRE0_EL1.STREn == '0' then
                if HCR_EL2.TGE == '0' then
                    GCSSTRTrapException(EL1);
                else
                    GCSSTRTrapException(EL2);
        when EL1
            if GCSCR_EL1.STREn == '0' then
                GCSSTRTrapException(EL1);
            elsif (EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
                   HFGITR_EL2.nGCSSTR_EL1 == '0') then
                GCSSTRTrapException(EL2);
        when EL2
            if GCSCR_EL2.STREn == '0' then
                GCSSTRTrapException(EL2);
        when EL3
            if GCSCR_EL3.STREn == '0' then
                GCSSTRTrapException(EL3);
    return;
```

Library pseudocode for aarch64/functions/gcs/EXLOCKException

```
// EXLOCKException()
// =================
// Handle an EXLOCK exception condition.

EXLOCKException()
    bits(64) preferred_exception_return = ThisInstrAddr(64);
    integer vect_offset = 0x0;

    except = ExceptionSyndrome(Exception_GCSFail);
    except.syndrome<24> = Zeros();
    except.syndrome<23:20> = '0001';
    except.syndrome<19:0> = Zeros();
    AArch64.TakeException(PSTATE.EL, except, preferred_exception_return, vect_offset);
```

Library pseudocode for aarch64/functions/gcs/GCSDataCheckException

```
// GCSDataCheckException()
// =======================
// Handle a Guarded Control Stack data check fault condition.

GCSDataCheckException(GCSInstruction gcsinst_type)
    bits(2) target_el;
    bits(64) preferred_exception_return = ThisInstrAddr(64);
    integer vect_offset = 0x0;
    boolean rn_unknown = FALSE;
    boolean is_ret = FALSE;

    if PSTATE.EL == EL0 then
        target_el = if HCR_EL2.TGE == '0' then EL1 else EL2;
    else
        target_el = PSTATE.EL;
    except = ExceptionSyndrome(Exception_GCSFail);
    case gcsinst_type of
        when GCSInstType_PRET
            except.syndrome<4:0> = '00000';
            is_ret = TRUE;
        when GCSInstType_POPM
            except.syndrome<4:0> = '00001';
        when GCSInstType_PRETAA
            except.syndrome<4:0> = '00010';
            is_ret = TRUE;
        when GCSInstType_PRETAB
            except.syndrome<4:0> = '00011';
            is_ret = TRUE;
        when GCSInstType_SS1
            except.syndrome<4:0> = '00100';
        when GCSInstType_SS2
            except.syndrome<4:0> = '00101';
            rn_unknown = TRUE;
        when GCSInstType_POPCX
            rn_unknown = TRUE;
            except.syndrome<4:0> = '01000';
        when GCSInstType_POPX
            except.syndrome<4:0> = '01001';
    if rn_unknown == TRUE then
        except.syndrome<9:5> = bits(5) UNKNOWN;
    elsif is_ret == TRUE then
        except.syndrome<9:5> = ThisInstr()<9:5>;
    else
        except.syndrome<9:5> = ThisInstr()<4:0>;
    except.syndrome<24:10> = Zeros();
    except.vaddress = bits(64) UNKNOWN;
    AArch64.TakeException(target_el, except, preferred_exception_return, vect_offset);
```

Library pseudocode for aarch64/functions/gcs/GCSEnabled

```
// GCSEnabled()
// ============
// Returns TRUE if the Guarded control stack is enabled at
// the provided Exception level.

boolean GCSEnabled(bits(2) el)
    if UsingAArch32() then
        return FALSE;

    if HaveEL(EL3) && el != EL3 && SCR_EL3.GCSEn == '0' then
        return FALSE;

    if (el IN {EL0, EL1} && EL2Enabled() &&
        (HCR_EL2.TGE == '0' || HCR_EL2.E2H == '0') &&
        (!IsHCRXEL2Enabled() || HCRX_EL2.GCSEn == '0')) then
        return FALSE;

    return GCSPCRSelected(el);
```

Library pseudocode for aarch64/functions/gcs/GCSInstruction

```
// GCSInstruction
// ==============

enumeration GCSInstruction {
    GCSInstType_PRET,    // Procedure return without Pointer authentication
    GCSInstType_POPM,    // GCSPOPM instruction
    GCSInstType_PRETAA,  // Procedure return with Pointer authentication that used key A
    GCSInstType_PRETAB,  // Procedure return with Pointer authentication that used key B
    GCSInstType_SS1,     // GCSSS1 instruction
    GCSInstType_SS2,     // GCSSS2 instruction
    GCSInstType_POPCX,   // GCSPOPCX instruction
    GCSInstType_POPX     // GCSPOPX instruction
};
```

Library pseudocode for aarch64/functions/gcs/GCSPCREnabled

```
// GCSPCREnabled()
// ===============
// Returns TRUE if the Guarded control stack is PCR enabled
// at the provided Exception level.

boolean GCSPCREnabled(bits(2) el)
    return GCSPCRSelected(el) && GCSEnabled(el);
```

Library pseudocode for aarch64/functions/gcs/GCSPCRSelected

```
// GCSPCRSelected()
// ================
// Returns TRUE if the Guarded control stack is PCR selected
// at the provided Exception level.

boolean GCSPCRSelected(bits(2) el)
    case el of
        when EL0 return GCSCRE0_EL1.PCRSEL == '1';
        when EL1 return GCSCR_EL1.PCRSEL == '1';
        when EL2 return GCSCR_EL2.PCRSEL == '1';
        when EL3 return GCSCR_EL3.PCRSEL == '1';
    Unreachable();
    return TRUE;
```

Library pseudocode for aarch64/functions/gcs/GCSPOPCX

```
// GCSPOPCX()
// ==========
// Called to pop and compare a Guarded control stack exception return record.

GCSPOPCX()
    bits(64) spsr = SPSR[];
    if !GCSEnabled(PSTATE.EL) then
        EndOfInstruction();
    CheckGCSExRecord(ELR[], spsr, X[30,64], GCSInstType_POPCX);
    PSTATE.EXLOCK = if GetCurrentEXLOCKEN() then '1' else '0';
    return;
```

Library pseudocode for aarch64/functions/gcs/GCSPOPM

```
// GCSPOPM()
// =========
// Called to pop a Guarded control stack procedure return record.

bits(64) GCSPOPM()
    bits(64) ptr;
    AccessDescriptor accdesc = CreateAccDescGCS(PSTATE.EL, MemOp_LOAD);

    if !GCSEnabled(PSTATE.EL) then EndOfInstruction();
    ptr = GetCurrentGCSPointer();
    bits(64) entry = Mem[ptr, 8, accdesc];

    if entry<1:0> != '00' then
        GCSDataCheckException(GCSInstType_POPM);

    ptr = ptr + 8;
    SetCurrentGCSPointer(ptr);
    return entry;
```

Library pseudocode for aarch64/functions/gcs/GCSPOPX

```
// GCSPOPX()
// =========
// Called to pop a Guarded control stack exception return record.

GCSPOPX()
    if !GCSEnabled(PSTATE.EL) then EndOfInstruction();

    bits(64) ptr;
    AccessDescriptor accdesc = CreateAccDescGCS(PSTATE.EL, MemOp_LOAD);
    ptr = GetCurrentGCSPointer();

    // Check the lowest doubleword is correctly formatted
    bits(64) recorded_first_dword = Mem[ptr, 8, accdesc];
    if recorded_first_dword != Zeros(60):'1001' then
        GCSDataCheckException(GCSInstType_POPX);

    // Ignore these loaded values, however they might have
    // faulted which is why we load them anyway
    bits(64) recorded_elr = Mem[ptr+8, 8, accdesc];
    bits(64) recorded_spsr = Mem[ptr+16, 8, accdesc];
    bits(64) recorded_lr = Mem[ptr+24, 8, accdesc];

    // Increment the pointer value
    ptr = ptr + 32;

    SetCurrentGCSPointer(ptr);
    return;
```

Library pseudocode for aarch64/functions/gcs/GCSPUSHM

```
// GCSPUSHM()
// ==========
// Called to push a Guarded control stack procedure return record.

GCSPUSHM(bits(64) value)
    if !GCSEnabled(PSTATE.EL) then EndOfInstruction();
    AddGCSRecord(value);
    return;
```

Library pseudocode for aarch64/functions/gcs/GCSPUSHX

```
// GCSPUSHX()
// ==========
// Called to push a Guarded control stack exception return record.

GCSPUSHX()
    bits(64) spsr = SPSR[];
    if !GCSEnabled(PSTATE.EL) then
        EndOfInstruction();
    AddGCSExRecord(ELR[], spsr, X[30,64]);
    PSTATE.EXLOCK = '0';
    return;
```

Library pseudocode for aarch64/functions/gcs/GCSReturnValueCheckEnabled

```
// GCSReturnValueCheckEnabled()
// ============================
// Returns TRUE if the Guarded control stack has return value
// checking enabled at the current Exception level.

boolean GCSReturnValueCheckEnabled(bits(2) el)
    if UsingAArch32() then
        return FALSE;
    case el of
        when EL0 return GCSCRE0_EL1.RVCHKEN == '1';
        when EL1 return GCSCR_EL1.RVCHKEN == '1';
        when EL2 return GCSCR_EL2.RVCHKEN == '1';
        when EL3 return GCSCR_EL3.RVCHKEN == '1';
```

Library pseudocode for aarch64/functions/gcs/GCSSS1

```
// GCSSS1()
// ========
// Operational pseudocode for GCSSS1 instruction.

GCSSS1(bits(64) incoming_pointer)
    bits(64) outgoing_pointer, cmpoperand, operand, data;
    if !GCSEnabled(PSTATE.EL) then EndOfInstruction();
    AccessDescriptor accdesc = CreateAccDescGCSSS1(PSTATE.EL);
    outgoing_pointer = GetCurrentGCSPointer();
    // Valid cap entry is expected
    cmpoperand = incoming_pointer[63:12]:'000000000001';
    // In-progress cap entry should be stored if the comparison is successful
    operand    = outgoing_pointer[63:3]:'101';

    data = MemAtomic(incoming_pointer, cmpoperand, operand, accdesc);
    if data == cmpoperand then
        SetCurrentGCSPointer(incoming_pointer[63:3]:'000');
    else
        GCSDataCheckException(GCSInstType_SS1);
    return;
```

Library pseudocode for aarch64/functions/gcs/GCSSS2

```
// GCSSS2()
// ========
// Operational pseudocode for GCSSS2 instruction.

bits(64) GCSSS2()
    bits(64) outgoing_pointer, incoming_pointer, outgoing_value;
    AccessDescriptor accdesc_ld = CreateAccDescGCS(PSTATE.EL, MemOp_LOAD);
    AccessDescriptor accdesc_st = CreateAccDescGCS(PSTATE.EL, MemOp_STORE);
    if !GCSEnabled(PSTATE.EL) then EndOfInstruction();
    incoming_pointer = GetCurrentGCSPointer();
    outgoing_value = Mem[incoming_pointer, 8, accdesc_ld];

    if outgoing_value[2:0] == '101' then  //in_progress token
        outgoing_pointer[63:3] = outgoing_value[63:3] - 1;
        outgoing_pointer[2:0] = '000';
        outgoing_value = outgoing_pointer[63:12]: '000000000001';
        Mem[outgoing_pointer, 8, accdesc_st] = outgoing_value;
        SetCurrentGCSPointer(incoming_pointer + 8);
        GCSSynchronizationBarrier();
    else
        GCSDataCheckException(GCSInstType_SS2);
    return outgoing_pointer;
```

Library pseudocode for aarch64/functions/gcs/GCSSTRTrapException

```
// GCSSTRTrapException()
// =====================
// Handle a trap on GCSSTR instruction condition.

GCSSTRTrapException(bits(2) target_el)
    bits(64) preferred_exception_return = ThisInstrAddr(64);
    integer vect_offset = 0x0;

    except = ExceptionSyndrome(Exception_GCSFail);
    except.syndrome<24> = Zeros();
    except.syndrome<23:20> = '0010';
    except.syndrome<19:15> = Zeros();
    except.syndrome<14:10> = ThisInstr()<9:5>;
    except.syndrome<9:5> = ThisInstr()<4:0>;
    except.syndrome<4:0> = Zeros();
    AArch64.TakeException(target_el, except, preferred_exception_return, vect_offset);
```

Library pseudocode for aarch64/functions/gcs/GCSSynchronizationBarrier

```
// GCSSynchronizationBarrier()
// ===========================
// Barrier instruction that synchronizes Guarded Control Stack
// accesses in relation to other load and store accesses

GCSSynchronizationBarrier();
```

Library pseudocode for aarch64/functions/gcs/GetCurrentEXLOCKEN

```
// GetCurrentEXLOCKEN()
// ====================

boolean GetCurrentEXLOCKEN()
    case PSTATE.EL of
        when EL0
            Unreachable();
        when EL1
            return GCSCR_EL1.EXLOCKEN == '1';
        when EL2
            return GCSCR_EL2.EXLOCKEN == '1';
        when EL3
            return GCSCR_EL3.EXLOCKEN == '1';
```

Library pseudocode for aarch64/functions/gcs/GetCurrentGCSPointer

```
// GetCurrentGCSPointer()
// ======================
// Returns the value of the current Guarded control stack
// pointer register.

bits(64) GetCurrentGCSPointer()
    bits(64) ptr;

    case PSTATE.EL of
        when EL0
            ptr = GCSPR_EL0.PTR:'000';
        when EL1
            ptr = GCSPR_EL1.PTR:'000';
        when EL2
            ptr = GCSPR_EL2.PTR:'000';
        when EL3
            ptr = GCSPR_EL3.PTR:'000';
    return ptr;
```

Library pseudocode for aarch64/functions/gcs/LoadCheckGCSRecord

```
// LoadCheckGCSRecord()
// ====================
// Validates the provided address against the top entry of the
// current Guarded control stack.

bits(64) LoadCheckGCSRecord(bits(64) vaddress, GCSInstruction gcsinst_type)
    bits(64) ptr;
    bits(64) recorded_va;
    AccessDescriptor accdesc = CreateAccDescGCS(PSTATE.EL, MemOp_LOAD);

    ptr = GetCurrentGCSPointer();
    recorded_va = Mem[ptr, 8, accdesc];
    if GCSReturnValueCheckEnabled(PSTATE.EL) && (recorded_va != vaddress) then
        GCSDataCheckException(gcsinst_type);

    return recorded_va;
```

Library pseudocode for aarch64/functions/gcs/SetCurrentGCSPointer

```
// SetCurrentGCSPointer()
// ======================
// Writes a value to the current Guarded control stack pointer register.

SetCurrentGCSPointer(bits(64) ptr)
    case PSTATE.EL of
        when EL0
            GCSPR_EL0.PTR = ptr<63:3>;
        when EL1
            GCSPR_EL1.PTR = ptr<63:3>;
        when EL2
            GCSPR_EL2.PTR = ptr<63:3>;
        when EL3
            GCSPR_EL3.PTR = ptr<63:3>;
    return;
```

Library pseudocode for aarch64/functions/ic/AArch64.IC

```
// AArch64.IC()
// ============
// Perform Instruction Cache Operation.

AArch64.IC(CacheOpScope opscope)
    regval = bits(64) UNKNOWN;
    AArch64.IC(regval, opscope);

// AArch64.IC()
// ============
// Perform Instruction Cache Operation.

AArch64.IC(bits(64) regval, CacheOpScope opscope)
    CacheRecord cache;

    cache.acctype   = AccessType_IC;
    cache.cachetype = CacheType_Instruction;
    cache.cacheop   = CacheOp_Invalidate;
    cache.opscope   = opscope;

    if opscope IN {CacheOpScope_ALLU, CacheOpScope_ALLUIS} then
        ss = SecurityStateAtEL(PSTATE.EL);
        cache.cpas = CPASAtSecurityState(ss);
        if (opscope == CacheOpScope_ALLUIS || (opscope == CacheOpScope_ALLU && PSTATE.EL == EL1
            && EL2Enabled() && HCR_EL2.FB == '1')) then
            cache.shareability = Shareability_ISH;
        else
            cache.shareability = Shareability_NSH;
        cache.regval = regval;
        CACHE_OP(cache);
    else
        assert opscope == CacheOpScope_PoU;

        if EL2Enabled() && !IsInHost() then
            if PSTATE.EL IN {EL0, EL1} then
                cache.is_vmid_valid = TRUE;
                cache.vmid          = VMID[];
            else
                cache.is_vmid_valid = FALSE;
        else
            cache.is_vmid_valid = FALSE;

        if PSTATE.EL == EL0 then
            cache.is_asid_valid = TRUE;
            cache.asid          = ASID[];
        else
            cache.is_asid_valid = FALSE;

        bits(64) vaddress = regval;
        boolean need_translate = ICInstNeedsTranslation(opscope);

        cache.vaddress     = regval;
        cache.shareability = Shareability_NSH;
        cache.translated   = need_translate;

        if !need_translate then
            cache.paddress = FullAddress UNKNOWN;
            CACHE_OP(cache);
            return;

        AccessDescriptor accdesc = CreateAccDescIC(cache);
        boolean aligned = TRUE;
        integer size = 0;
        AddressDescriptor memaddrdesc = AArch64.TranslateAddress(vaddress, accdesc, aligned, size);

        if IsFault(memaddrdesc) then
            AArch64.Abort(regval, memaddrdesc.fault);

        cache.cpas     = CPASAtPAS(memaddrdesc.paddress.paspace);
        cache.paddress = memaddrdesc.paddress;
```

```
        CACHE_OP(cache);
    return;
```

Library pseudocode for aarch64/functions/immediateop/ImmediateOp

```
// ImmediateOp
// ===========
// Vector logical immediate instruction types.

enumeration ImmediateOp {ImmediateOp_MOVI, ImmediateOp_MVNI,
                         ImmediateOp_ORR, ImmediateOp_BIC};
```

Library pseudocode for aarch64/functions/logicalop/LogicalOp

```
// LogicalOp
// =========
// Logical instruction types.

enumeration LogicalOp   {LogicalOp_AND, LogicalOp_EOR, LogicalOp_ORR};
```

Library pseudocode for aarch64/functions/mec/AArch64.S1AMECFault

```
// AArch64.S1AMECFault()
// =====================
// Returns TRUE if a Translation fault should occur for Realm EL2 and Realm EL2&0
// stage 1 translated addresses to Realm PA space.

boolean AArch64.S1AMECFault(S1TTWParams walkparams, PASpace paspace, Regime regime,
                            bits(N) descriptor)
    assert N IN {64,128};
    bit descriptor_amec = if walkparams.d128 == '1' then descriptor<103> else descriptor<63>;

    return (walkparams.<emec,amec> == '10' &&
            regime IN {Regime_EL2, Regime_EL20} &&
            paspace == PAS_Realm &&
            descriptor_amec == '1');
```

Library pseudocode for aarch64/functions/mec/AArch64.S1DisabledOutputMECID

```
// AArch64.S1DisabledOutputMECID()
// ===============================
// Returns the output MECID when stage 1 address translation is disabled.

bits(16) AArch64.S1DisabledOutputMECID(S1TTWParams walkparams, Regime regime, PASpace paspace)
    if walkparams.emec == '0' then
        return DEFAULT_MECID;

    if !(regime IN {Regime_EL2, Regime_EL20, Regime_EL10}) then
        return DEFAULT_MECID;

    if paspace != PAS_Realm then
        return DEFAULT_MECID;

    if regime == Regime_EL10 then
        return VMECID_P_EL2.MECID;
    else
        return MECID_P0_EL2.MECID;
```

Library pseudocode for aarch64/functions/mec/AArch64.S1OutputMECID

```
// AArch64.S1OutputMECID()
// =======================
// Returns the output MECID when stage 1 address translation is enabled.

bits(16) AArch64.S1OutputMECID(S1TTWParams walkparams, Regime regime, VARange varange,
                               PASpace paspace, bits(N) descriptor)
    assert N IN {64,128};

    if walkparams.emec == '0' then
        return DEFAULT_MECID;

    if paspace != PAS_Realm then
        return DEFAULT_MECID;

    bit descriptor_amec = if walkparams.d128 == '1' then descriptor<103> else descriptor<63>;
    case regime of
        when Regime_EL3
            return MECID_RL_A_EL3.MECID;
        when Regime_EL2
            if descriptor_amec == '0' then
                return MECID_P0_EL2.MECID;
            else
                return MECID_A0_EL2.MECID;
        when Regime_EL20
            if varange == VARange_LOWER then
                if descriptor_amec == '0' then
                    return MECID_P0_EL2.MECID;
                else
                    return MECID_A0_EL2.MECID;
            else
                if descriptor_amec == '0' then
                    return MECID_P1_EL2.MECID;
                else
                    return MECID_A1_EL2.MECID;
        when Regime_EL10
            return VMECID_P_EL2.MECID;
```

Library pseudocode for aarch64/functions/mec/AArch64.S2OutputMECID

```
// AArch64.S2OutputMECID()
// =======================
// Returns the output MECID for stage 2 address translation.

bits(16) AArch64.S2OutputMECID(S2TTWParams walkparams, PASpace paspace, bits(N) descriptor)
    assert N IN {64,128};

    if walkparams.emec == '0' then
        return DEFAULT_MECID;

    if paspace != PAS_Realm then
        return DEFAULT_MECID;

    bit descriptor_amec = if walkparams.d128 == '1' then descriptor<103> else descriptor<63>;
    if descriptor_amec == '0' then
        return VMECID_P_EL2.MECID;
    else
        return VMECID_A_EL2.MECID;
```

Library pseudocode for aarch64/functions/mec/AArch64.TTWalkMECID

```
// AArch64.TTWalkMECID()
// =====================
// Returns the associated MECID for the translation table walk of the given
// translation regime and Security state.

bits(16) AArch64.TTWalkMECID(bit emec, Regime regime, SecurityState ss)
    if emec == '0' then
        return DEFAULT_MECID;

    if ss != SS_Realm then
        return DEFAULT_MECID;

    case regime of
        when Regime_EL2
            return MECID_P0_EL2.MECID;
        when Regime_EL20
            if TCR_EL2.A1 == '0' then
                return MECID_P1_EL2.MECID;
            else
                return MECID_P0_EL2.MECID;
        // This applies to stage 1 and stage 2 translation table walks for
        // Realm EL1&0, but the stage 2 translation for a stage 1 walk
        // might later override the MECID according to AMEC configuration.
        when Regime_EL10
            return VMECID_P_EL2.MECID;
        otherwise
            Unreachable();
```

Library pseudocode for aarch64/functions/mec/DEFAULT_MECID

```
constant bits(16) DEFAULT_MECID = Zeros(16);
```

Library pseudocode for aarch64/functions/memory/AArch64.AccessIsTagChecked

```
// AArch64.AccessIsTagChecked()
// ============================
// TRUE if a given access is tag-checked, FALSE otherwise.

boolean AArch64.AccessIsTagChecked(bits(64) vaddr, AccessDescriptor accdesc)
    assert accdesc.tagchecked;

    if UsingAArch32() then
        return FALSE;

    boolean is_instr = FALSE;
    if (EffectiveMTX(vaddr, is_instr, PSTATE.EL) == '0' &&
        EffectiveTBI(vaddr, is_instr, PSTATE.EL) == '0') then
        return FALSE;

    if (EffectiveTCMA(vaddr, PSTATE.EL) == '1' &&
            (vaddr<59:55> == '00000' || vaddr<59:55> == '11111')) then
        return FALSE;

    if !AArch64.AllocationTagAccessIsEnabled(accdesc.el) then
        return FALSE;

    if PSTATE.TCO=='1' then
        return FALSE;

    if HaveMTEStoreOnlyExt() && !accdesc.write && StoreOnlyTagCheckingEnabled(accdesc.el) then
        return FALSE;

    return TRUE;
```

Library pseudocode for aarch64/functions/memory/AArch64.AddressWithAllocationTag

```
// AArch64.AddressWithAllocationTag()
// ==================================
// Generate a 64-bit value containing a Logical Address Tag from a 64-bit
// virtual address and an Allocation Tag.
// If the extension is disabled, treats the Allocation Tag as '0000'.

bits(64) AArch64.AddressWithAllocationTag(bits(64) address, bits(4) allocation_tag)
    bits(64) result = address;
    bits(4) tag;
    if AArch64.AllocationTagAccessIsEnabled(PSTATE.EL) then
        tag = allocation_tag;
    else
        tag = '0000';
    result<59:56> = tag;
    return result;
```

Library pseudocode for aarch64/functions/memory/AArch64.AllocationTagCheck

```
// AArch64.AllocationTagCheck()
// ============================
// Performs an Allocation Tag Check operation for a memory access and
// returns whether the check passed.

boolean AArch64.AllocationTagCheck(AddressDescriptor memaddrdesc, AccessDescriptor accdesc,
                                   bits(4) ptag)
    if memaddrdesc.memattrs.tags == MemTag_AllocationTagged then
        (memstatus, readtag) = PhysMemTagRead(memaddrdesc, accdesc);
        if IsFault(memstatus) then
            HandleExternalReadAbort(memstatus, memaddrdesc, 1, accdesc);

        return ptag == readtag;
    else
        return TRUE;
```

Library pseudocode for aarch64/functions/memory/AArch64.AllocationTagFromAddress

```
// AArch64.AllocationTagFromAddress()
// ==================================
// Generate an Allocation Tag from a 64-bit value containing a Logical Address Tag.

bits(4) AArch64.AllocationTagFromAddress(bits(64) tagged_address)
    return tagged_address<59:56>;
```

Library pseudocode for aarch64/functions/memory/AArch64.CanonicalTagCheck

```
// AArch64.CanonicalTagCheck()
// ===========================
// Performs a Canonical Tag Check operation for a memory access and
// returns whether the check passed.

boolean AArch64.CanonicalTagCheck(AddressDescriptor memaddrdesc, bits(4) ptag)
    expected_tag = if memaddrdesc.vaddress<55> == '0' then '0000' else '1111';
    return ptag == expected_tag;
```

Library pseudocode for aarch64/functions/memory/AArch64.CheckTag

```
// AArch64.CheckTag()
// ==================
// Performs a Tag Check operation for a memory access and returns
// whether the check passed

boolean AArch64.CheckTag(AddressDescriptor memaddrdesc, AccessDescriptor accdesc, bits(4) ptag)
    if memaddrdesc.memattrs.tags == MemTag_AllocationTagged then
        return AArch64.AllocationTagCheck(memaddrdesc, accdesc, ptag);
    elsif memaddrdesc.memattrs.tags == MemTag_CanonicallyTagged then
        return AArch64.CanonicalTagCheck(memaddrdesc, ptag);
    else
        return TRUE;
```

Library pseudocode for aarch64/functions/memory/AArch64.IsUnprivAccessPriv

```
// AArch64.IsUnprivAccessPriv()
// ============================
// Returns TRUE if an unprivileged access is privileged, and FALSE otherwise.

boolean AArch64.IsUnprivAccessPriv()
    boolean privileged;

    case PSTATE.EL of
        when EL0 privileged = FALSE;
        when EL1 privileged = EL2Enabled() && HaveNVExt() && HCR_EL2.<NV,NV1> == '11';
        when EL2 privileged = !(HaveVirtHostExt() && HCR_EL2.<E2H,TGE> == '11');
        when EL3 privileged = TRUE;

    if HaveUAOExt() && PSTATE.UAO == '1' then
        privileged = PSTATE.EL != EL0;

    return privileged;
```

Library pseudocode for aarch64/functions/memory/AArch64.MemSingle

```
// AArch64.MemSingle[] - non-assignment (read) form
// ================================================
// Perform an atomic, little-endian read of 'size' bytes.

bits(size*8) AArch64.MemSingle[bits(64) address, integer size,
                               AccessDescriptor accdesc, boolean aligned]
    boolean ispair = FALSE;
    return AArch64.MemSingle[address, size, accdesc, aligned, ispair];

// AArch64.MemSingle[] - non-assignment (read) form
// ================================================
// Perform an atomic, little-endian read of 'size' bytes.

bits(size*8) AArch64.MemSingle[bits(64) address, integer size, AccessDescriptor accdesc_in,
                               boolean aligned, boolean ispair]
    assert size IN {1, 2, 4, 8, 16};
    bits(size*8) value;
    AccessDescriptor accdesc = accdesc_in;
    if HaveLSE2Ext() then
        assert AllInAlignedQuantity(address, size, 16);
    else
        assert IsAligned(address, size);

    // If the instruction encoding permits tag checking, confer with system register configuration
    // which may override this.
    if HaveMTE2Ext() && accdesc.tagchecked then
        accdesc.tagchecked = AArch64.AccessIsTagChecked(address, accdesc);

    AddressDescriptor memaddrdesc;
    memaddrdesc = AArch64.TranslateAddress(address, accdesc, aligned, size);

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        AArch64.Abort(address, memaddrdesc.fault);

    // Memory array access
    if HaveTME() then
        if accdesc.transactional && !MemHasTransactionalAccess(memaddrdesc.memattrs) then
            FailTransaction(TMFailure_IMP, FALSE);

    if HaveMTE2Ext() && accdesc.tagchecked then
        bits(4) ptag = AArch64.PhysicalTag(address);
        if !AArch64.CheckTag(memaddrdesc, accdesc, ptag) then
            AArch64.TagCheckFault(address, accdesc);

    if SPESampleInFlight then
        boolean is_load = TRUE;
        SPESampleLoadStore(is_load, accdesc, memaddrdesc);

    boolean atomic;
    if (memaddrdesc.memattrs.memtype == MemType_Normal &&
            memaddrdesc.memattrs.inner.attrs == MemAttr_WB &&
            memaddrdesc.memattrs.outer.attrs == MemAttr_WB) then
        atomic = TRUE;
    elsif (accdesc.exclusive || accdesc.atomicop ||
            accdesc.acqsc || accdesc.acqpc || accdesc.relsc) then
        if !aligned && !ConstrainUnpredictableBool(Unpredictable_MISALIGNEDATOMIC) then
            AArch64.Abort(address, AlignmentFault(accdesc));
        else
            atomic = TRUE;
    elsif aligned then
        atomic = !ispair;
    else
        // Misaligned accesses within 16 byte aligned memory but
        // not Normal Cacheable Writeback are Atomic
        atomic = boolean IMPLEMENTATION_DEFINED "FEAT_LSE2: access is atomic";

    PhysMemRetStatus memstatus;
    if atomic then
        (memstatus, value) = PhysMemRead(memaddrdesc, size, accdesc);
```

```
        if IsFault(memstatus) then
            HandleExternalReadAbort(memstatus, memaddrdesc, size, accdesc);
    elsif aligned && ispair then
        assert size IN {8, 16};
        constant halfsize = size DIV 2;
        bits(halfsize * 8) lowhalf, highhalf;
        (memstatus, lowhalf) = PhysMemRead(memaddrdesc, halfsize, accdesc);
        if IsFault(memstatus) then
            HandleExternalReadAbort(memstatus, memaddrdesc, halfsize, accdesc);
        memaddrdesc.paddress.address = memaddrdesc.paddress.address + halfsize;
        (memstatus, highhalf) = PhysMemRead(memaddrdesc, halfsize, accdesc);
        if IsFault(memstatus) then
            HandleExternalReadAbort(memstatus, memaddrdesc, halfsize, accdesc);

        value = highhalf:lowhalf;
    else
        for i = 0 to size-1
            (memstatus, value<8*i+7:8*i>) = PhysMemRead(memaddrdesc, 1, accdesc);
            if IsFault(memstatus) then
                HandleExternalReadAbort(memstatus, memaddrdesc, 1, accdesc);
            memaddrdesc.paddress.address = memaddrdesc.paddress.address + 1;
    return value;

// AArch64.MemSingle[] - assignment (write) form
// =============================================

AArch64.MemSingle[bits(64) address, integer size,
                  AccessDescriptor accdesc, boolean aligned] = bits(size*8) value
    boolean ispair = FALSE;
    AArch64.MemSingle[address, size, accdesc, aligned, ispair] = value;
    return;

// AArch64.MemSingle[] - assignment (write) form
// =============================================
// Perform an atomic, little-endian write of 'size' bytes.

AArch64.MemSingle[bits(64) address, integer size, AccessDescriptor accdesc_in,
                  boolean aligned, boolean ispair] = bits(size*8) value
    assert size IN {1, 2, 4, 8, 16};
    AccessDescriptor accdesc = accdesc_in;
    if HaveLSE2Ext() then
        assert AllInAlignedQuantity(address, size, 16);
    else
        assert IsAligned(address, size);

    // If the instruction encoding permits tag checking, confer with system register configuration
    // which may override this.
    if HaveMTE2Ext() && accdesc.tagchecked then
        accdesc.tagchecked = AArch64.AccessIsTagChecked(address, accdesc);

    AddressDescriptor memaddrdesc;
    memaddrdesc = AArch64.TranslateAddress(address, accdesc, aligned, size);

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        AArch64.Abort(address, memaddrdesc.fault);

    // Effect on exclusives
    if memaddrdesc.memattrs.shareability != Shareability_NSH then
        ClearExclusiveByAddress(memaddrdesc.paddress, ProcessorID(), size);

    if HaveTME() then
        if accdesc.transactional && !MemHasTransactionalAccess(memaddrdesc.memattrs) then
            FailTransaction(TMFailure_IMP, FALSE);

    if HaveMTE2Ext() && accdesc.tagchecked then
        bits(4) ptag = AArch64.PhysicalTag(address);
        if !AArch64.CheckTag(memaddrdesc, accdesc, ptag) then
            AArch64.TagCheckFault(address, accdesc);
```

```
    if SPESampleInFlight then
        boolean is_load = FALSE;
        SPESampleLoadStore(is_load, accdesc, memaddrdesc);

    PhysMemRetStatus memstatus;
    boolean atomic;
    if (memaddrdesc.memattrs.memtype == MemType_Normal &&
            memaddrdesc.memattrs.inner.attrs == MemAttr_WB &&
            memaddrdesc.memattrs.outer.attrs == MemAttr_WB) then
        atomic = TRUE;
    elsif (accdesc.exclusive || accdesc.atomicop ||
            accdesc.acqsc || accdesc.acqpc || accdesc.relsc) then
        if !aligned && !ConstrainUnpredictableBool(Unpredictable_MISALIGNEDATOMIC) then
            AArch64.Abort(address, AlignmentFault(accdesc));
        else
            atomic = TRUE;
    elsif aligned then
        atomic = !ispair;
    else
        // Misaligned accesses within 16 byte aligned memory but
        // not Normal Cacheable Writeback are Atomic
        atomic = boolean IMPLEMENTATION_DEFINED "FEAT_LSE2: access is atomic";

    if atomic then
        memstatus = PhysMemWrite(memaddrdesc, size, accdesc, value);
        if IsFault(memstatus) then
            HandleExternalWriteAbort(memstatus, memaddrdesc, size, accdesc);
    elsif aligned && ispair then
        assert size IN {8, 16};
        constant halfsize = size DIV 2;
        bits(halfsize*8) lowhalf, highhalf;
        <highhalf, lowhalf> = value;

        memstatus = PhysMemWrite(memaddrdesc, halfsize, accdesc, lowhalf);
        if IsFault(memstatus) then
            HandleExternalWriteAbort(memstatus, memaddrdesc, halfsize, accdesc);
        memaddrdesc.paddress.address = memaddrdesc.paddress.address + halfsize;
        memstatus = PhysMemWrite(memaddrdesc, halfsize, accdesc, highhalf);
        if IsFault(memstatus) then
            HandleExternalWriteAbort(memstatus, memaddrdesc, halfsize, accdesc);
    else
        for i = 0 to size-1
            memstatus = PhysMemWrite(memaddrdesc, 1, accdesc, value<8*i+7:8*i>);
            if IsFault(memstatus) then
                HandleExternalWriteAbort(memstatus, memaddrdesc, 1, accdesc);
            memaddrdesc.paddress.address = memaddrdesc.paddress.address + 1;
    return;
```

Library pseudocode for aarch64/functions/memory/AArch64.MemTag

```
// AArch64.MemTag[] - non-assignment (read) form
// ============================================
// Load an Allocation Tag from memory.

bits(4) AArch64.MemTag[bits(64) address, AccessDescriptor accdesc_in]
    assert accdesc_in.tagaccess && !accdesc_in.tagchecked;

    AddressDescriptor memaddrdesc;
    AccessDescriptor accdesc = accdesc_in;

    boolean aligned = TRUE;

    if HaveMTE2Ext() then
        accdesc.tagaccess = AArch64.AllocationTagAccessIsEnabled(accdesc.el);

    memaddrdesc = AArch64.TranslateAddress(address, accdesc, aligned, TAG_GRANULE);

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        AArch64.Abort(address, memaddrdesc.fault);

    // Return the granule tag if tagging is enabled...
    if accdesc.tagaccess && memaddrdesc.memattrs.tags == MemTag_AllocationTagged then
        (memstatus, tag) = PhysMemTagRead(memaddrdesc, accdesc);
        if IsFault(memstatus) then
            HandleExternalReadAbort(memstatus, memaddrdesc, 1, accdesc);
        return tag;
    elsif (HaveMTECanonicalTagCheckingExt() &&
           accdesc.tagaccess &&
           memaddrdesc.memattrs.tags == MemTag_CanonicallyTagged) then
        return if address<55> == '0' then '0000' else '1111';
    else
        // ...otherwise read tag as zero.
        return '0000';

// AArch64.MemTag[] - assignment (write) form
// ==========================================
// Store an Allocation Tag to memory.

AArch64.MemTag[bits(64) address, AccessDescriptor accdesc_in] = bits(4) value
    assert accdesc_in.tagaccess && !accdesc_in.tagchecked;

    AddressDescriptor memaddrdesc;
    AccessDescriptor accdesc = accdesc_in;

    boolean aligned = IsAligned(address, TAG_GRANULE);

    // Stores of allocation tags must be aligned
    if !aligned then
        AArch64.Abort(address, AlignmentFault(accdesc));

    if HaveMTE2Ext() then
        accdesc.tagaccess = AArch64.AllocationTagAccessIsEnabled(accdesc.el);

    memaddrdesc = AArch64.TranslateAddress(address, accdesc, aligned, TAG_GRANULE);

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        AArch64.Abort(address, memaddrdesc.fault);

    // Memory array access
    if accdesc.tagaccess && memaddrdesc.memattrs.tags == MemTag_AllocationTagged then
        memstatus = PhysMemTagWrite(memaddrdesc, accdesc, value);
        if IsFault(memstatus) then
            HandleExternalWriteAbort(memstatus, memaddrdesc, 1, accdesc);
```

Library pseudocode for aarch64/functions/memory/AArch64.PhysicalTag

```
// AArch64.PhysicalTag()
// =====================
// Generate a Physical Tag from a Logical Tag in an address

bits(4) AArch64.PhysicalTag(bits(64) vaddr)
    return vaddr<59:56>;
```

Library pseudocode for aarch64/functions/memory/AArch64.UnalignedAccessFaults

```
// AArch64.UnalignedAccessFaults()
// ===============================
// Determine whether the unaligned access generates an Alignment fault

boolean AArch64.UnalignedAccessFaults(AccessDescriptor accdesc, bits(64) address, integer size)
    if AlignmentEnforced() then
        return TRUE;
    elsif accdesc.acctype == AccessType_GCS then
        return TRUE;
    elsif accdesc.rcw then
        return TRUE;
    elsif accdesc.ls64 then
        return TRUE;
    elsif accdesc.exclusive || accdesc.atomicop then
        return !HaveLSE2Ext() || !AllInAlignedQuantity(address, size, 16);
    elsif accdesc.acqsc || accdesc.acqpc || accdesc.relsc then
        return !HaveLSE2Ext() || (SCTLR[].nAA == '0' && !AllInAlignedQuantity(address, size, 16));
    else
        return FALSE;
```

Library pseudocode for aarch64/functions/memory/AddressSupportsLS64

```
// AddressSupportsLS64()
// =====================
// Returns TRUE if the 64-byte block following the given address supports the
// LD64B and ST64B instructions, and FALSE otherwise.

boolean AddressSupportsLS64(bits(56) paddress);
```

Library pseudocode for aarch64/functions/memory/AllInAlignedQuantity

```
// AllInAlignedQuantity()
// ======================
// Returns TRUE if all accessed bytes are within one aligned quantity, FALSE otherwise.

boolean AllInAlignedQuantity(bits(64) address, integer size, integer alignment)
    assert(size <= alignment);
    return Align((address+size)-1, alignment) == Align(address, alignment);
```

Library pseudocode for aarch64/functions/memory/CheckSPAlignment

```
// CheckSPAlignment()
// ==================
// Check correct stack pointer alignment for AArch64 state.

CheckSPAlignment()
    bits(64) sp = SP[];
    boolean stack_align_check;
    if PSTATE.EL == EL0 then
        stack_align_check = (SCTLR[].SA0 != '0');
    else
        stack_align_check = (SCTLR[].SA != '0');

    if stack_align_check && sp != Align(sp, 16) then
        AArch64.SPAlignmentFault();

    return;
```

Library pseudocode for aarch64/functions/memory/Mem

```
// Mem[] - non-assignment (read) form
// ==================================
// Perform a read of 'size' bytes. The access byte order is reversed for a big-endian access.
// Instruction fetches would call AArch64.MemSingle directly.

bits(size*8) Mem[bits(64) address, integer size, AccessDescriptor accdesc]
    boolean ispair = FALSE;
    boolean highestAddressfirst = FALSE;
    return Mem[address, size, accdesc, ispair, highestAddressfirst];

bits(size*8) Mem[bits(64) address, integer size, AccessDescriptor accdesc, boolean ispair]
    boolean highestAddressfirst = FALSE;
    return Mem[address, size, accdesc, ispair, highestAddressfirst];

bits(size*8) Mem[bits(64) address, integer size, AccessDescriptor accdesc,
                 boolean ispair, boolean  highestAddressfirst]
    assert size IN {1, 2, 4, 8, 16};
    constant halfsize = size DIV 2;
    bits(size * 8) value;
    bits(halfsize * 8) lowhalf, highhalf;

    // Check alignment on size of element accessed, not overall access size
    integer alignment = if ispair then halfsize else size;
    boolean aligned   = IsAligned(address, alignment);

    if !aligned && AArch64.UnalignedAccessFaults(accdesc, address, size) then
        AArch64.Abort(address, AlignmentFault(accdesc));

    if accdesc.acctype == AccessType_ASIMD && size == 16 && IsAligned(address, 8) then
        // If 128-bit SIMD&FP ordered access are treated as a pair of
        // 64-bit single-copy atomic accesses, then these single copy atomic
        // access can be observed in any order.
        lowhalf = AArch64.MemSingle[address, halfsize, accdesc, aligned, ispair];
        highhalf = AArch64.MemSingle[address+halfsize, halfsize, accdesc, aligned, ispair];
        value = highhalf:lowhalf;
    elsif HaveLSE2Ext() && AllInAlignedQuantity(address, size, 16) then
        value = AArch64.MemSingle[address, size, accdesc, aligned, ispair];
    elsif ispair && aligned then
        if HaveLRCPC3Ext() && highestAddressfirst then
            highhalf = AArch64.MemSingle[address+halfsize, halfsize, accdesc, aligned];
            lowhalf  = AArch64.MemSingle[address, halfsize, accdesc, aligned];
        else
            lowhalf  = AArch64.MemSingle[address, halfsize, accdesc, aligned];
            highhalf = AArch64.MemSingle[address+halfsize, halfsize, accdesc, aligned];
        value = highhalf:lowhalf;
    elsif aligned then
        value = AArch64.MemSingle[address, size, accdesc, aligned, ispair];
    else
        assert size > 1;
        if HaveLRCPC3Ext() && ispair && highestAddressfirst then
            // Performing memory accesses from one load or store instruction to Device memory that
            // crosses a boundary corresponding to the smallest translation granule size of the
            // implementation causes CONSTRAINED UNPREDICTABLE behavior.

            for i = 0 to halfsize-1
                // Individual byte access can be observed in any order
                highhalf<8*i+7:8*i> = AArch64.MemSingle[address+halfsize +i, 1, accdesc, aligned];
            for i = 0 to halfsize-1
                // Individual byte access can be observed in any order
                lowhalf<8*i+7:8*i> = AArch64.MemSingle[address + i, 1, accdesc, aligned];

            value = highhalf:lowhalf;

        else
            value<7:0> = AArch64.MemSingle[address, 1, accdesc, aligned];

            // For subsequent bytes it is CONSTRAINED UNPREDICTABLE whether an unaligned Device
            // memory access will generate an Alignment Fault, as to get this far means the first
            // byte did not, so we must be changing to a new translation page.
            c = ConstrainUnpredictable(Unpredictable_DEVPAGE2);
```

```
            assert c IN {Constraint_FAULT, Constraint_NONE};
            if c == Constraint_NONE then aligned = TRUE;

            for i = 1 to size-1
                value<8*i+7:8*i> = AArch64.MemSingle[address+i, 1, accdesc, aligned];

    if BigEndian(accdesc.acctype) then
        value = BigEndianReverse(value);

    return value;

// Mem[] - assignment (write) form
// ===============================
// Perform a write of 'size' bytes. The byte order is reversed for a big-endian access.

Mem[bits(64) address, integer size, AccessDescriptor accdesc] = bits(size*8) value_in
    boolean ispair = FALSE;
    boolean highestAddressfirst = FALSE;
    Mem[address, size, accdesc, ispair, highestAddressfirst] = value_in;

Mem[bits(64) address, integer size, AccessDescriptor accdesc,
        boolean ispair] = bits(size*8) value_in
    boolean highestAddressfirst = FALSE;
    Mem[address, size, accdesc, ispair, highestAddressfirst] = value_in;

Mem[bits(64) address, integer size, AccessDescriptor accdesc,
        boolean ispair, boolean highestAddressfirst] = bits(size*8) value_in
    constant halfsize = size DIV 2;
    bits(size*8) value = value_in;
    bits(halfsize*8) lowhalf, highhalf;

    // Check alignment on size of element accessed, not overall access size
    integer alignment = if ispair then halfsize else size;
    boolean aligned   = IsAligned(address, alignment);

    if !aligned && AArch64.UnalignedAccessFaults(accdesc, address, size) then
        AArch64.Abort(address, AlignmentFault(accdesc));

    if BigEndian(accdesc.acctype) then
        value = BigEndianReverse(value);

    if accdesc.acctype == AccessType_ASIMD && size == 16 && IsAligned(address, 8) then
        // 128-bit SIMD&FP stores are treated as a pair of 64-bit single-copy atomic accesses
        // 64-bit aligned.
        <highhalf, lowhalf> = value;
        AArch64.MemSingle[address, halfsize, accdesc, aligned, ispair] = lowhalf;
        AArch64.MemSingle[address+halfsize, halfsize, accdesc, aligned, ispair] = highhalf;
    elsif HaveLSE2Ext() && AllInAlignedQuantity(address, size, 16) then
        AArch64.MemSingle[address, size, accdesc, aligned, ispair] = value;
    elsif ispair && aligned then
        joinedpair = FALSE;
        <highhalf, lowhalf> = value;
        if HaveLRCPC3Ext() && highestAddressfirst then
            AArch64.MemSingle[address+halfsize, halfsize, accdesc, aligned, joinedpair] = highhalf;
            AArch64.MemSingle[address, halfsize, accdesc, aligned, joinedpair] = lowhalf;
        else
            AArch64.MemSingle[address, halfsize, accdesc, aligned, joinedpair] = lowhalf;
            AArch64.MemSingle[address+halfsize, halfsize, accdesc, aligned, joinedpair] = highhalf;
    elsif aligned then
        AArch64.MemSingle[address, size, accdesc, aligned, ispair] = value;
    else
        assert size > 1;
        if HaveLRCPC3Ext() && ispair && highestAddressfirst then
            // Performing memory accesses from one load or store instruction to Device memory that
            // crosses a boundary corresponding to the smallest translation granule size of the
            // implementation causes CONSTRAINED UNPREDICTABLE behavior.
            <highhalf, lowhalf> = value;
            for i = 0 to halfsize-1
                // Individual byte access can be observed in any order
                AArch64.MemSingle[address+halfsize+i, 1, accdesc, aligned] = highhalf<8*i+7:8*i>;
```

```
            for i = 0 to halfsize-1
                // Individual byte access can be observed in any order, but implies observability
                // of highhalf
                AArch64.MemSingle[address+i, 1, accdesc, aligned] = lowhalf<8*i+7:8*i>;
        else
            AArch64.MemSingle[address, 1, accdesc, aligned] = value<7:0>;

            // For subsequent bytes it is CONSTRAINED UNPREDICTABLE whether an unaligned Device
            // memory access will generate an Alignment Fault, as to get this far means the first
            // byte did not, so we must be changing to a new translation page.

            c = ConstrainUnpredictable(Unpredictable_DEVPAGE2);
            assert c IN {Constraint_FAULT, Constraint_NONE};
            if c == Constraint_NONE then aligned = TRUE;

            for i = 1 to size-1
                AArch64.MemSingle[address+i, 1, accdesc, aligned] = value<8*i+7:8*i>;
    return;
```

Library pseudocode for aarch64/functions/memory/MemAtomic

```
// MemAtomic()
// ===========
// Performs load and store memory operations for a given virtual address.

bits(size) MemAtomic(bits(64) address, bits(size) cmpoperand, bits(size) operand,
                     AccessDescriptor accdesc_in)
    assert accdesc_in.atomicop;

    constant integer bytes = size DIV 8;
    assert bytes IN {1, 2, 4, 8, 16};

    bits(size) newvalue;
    bits(size) oldvalue;
    AccessDescriptor accdesc = accdesc_in;
    boolean aligned = IsAligned(address, bytes);

    // If the instruction encoding permits tag checking, confer with system register configuration
    // which may override this.
    if HaveMTE2Ext() && accdesc.tagchecked then
        accdesc.tagchecked = AArch64.AccessIsTagChecked(address, accdesc);

    if !aligned && AArch64.UnalignedAccessFaults(accdesc, address, bytes) then
        AArch64.Abort(address, AlignmentFault(accdesc));

    // MMU or MPU lookup
    AddressDescriptor memaddrdesc = AArch64.TranslateAddress(address, accdesc, aligned, size);

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        AArch64.Abort(address, memaddrdesc.fault);

    // Effect on exclusives
    if memaddrdesc.memattrs.shareability != Shareability_NSH then
        ClearExclusiveByAddress(memaddrdesc.paddress, ProcessorID(), size);

    // For Store-only Tag checking, the tag check is performed on the store.
    if (HaveMTE2Ext() && accdesc.tagchecked &&
        (!HaveMTEStoreOnlyExt() || !StoreOnlyTagCheckingEnabled(accdesc.el))) then
        bits(4) ptag = AArch64.PhysicalTag(address);
        if !AArch64.CheckTag(memaddrdesc, accdesc, ptag) then
            AArch64.TagCheckFault(address, accdesc);

    // All observers in the shareability domain observe the following load and store atomically.
    PhysMemRetStatus memstatus;
    (memstatus, oldvalue) = PhysMemRead(memaddrdesc, bytes, accdesc);

    if IsFault(memstatus) then
        HandleExternalReadAbort(memstatus, memaddrdesc, bytes, accdesc);
    if BigEndian(accdesc.acctype) then
        oldvalue = BigEndianReverse(oldvalue);

    boolean cmpfail = FALSE;
    case accdesc.modop of
        when MemAtomicOp_ADD    newvalue = oldvalue + operand;
        when MemAtomicOp_BIC    newvalue = oldvalue AND NOT(operand);
        when MemAtomicOp_EOR    newvalue = oldvalue EOR operand;
        when MemAtomicOp_ORR    newvalue = oldvalue OR operand;
        when MemAtomicOp_SMAX   newvalue = Max(SInt(oldvalue), SInt(operand))<size-1:0>;
        when MemAtomicOp_SMIN   newvalue = Min(SInt(oldvalue), SInt(operand))<size-1:0>;
        when MemAtomicOp_UMAX   newvalue = Max(UInt(oldvalue), UInt(operand))<size-1:0>;
        when MemAtomicOp_UMIN   newvalue = Min(UInt(oldvalue), UInt(operand))<size-1:0>;
        when MemAtomicOp_SWP    newvalue = operand;
        when MemAtomicOp_CAS    newvalue = operand; cmpfail = cmpoperand != oldvalue;
        when MemAtomicOp_GCSSS1 newvalue = operand; cmpfail = cmpoperand != oldvalue;

    if HaveMTEStoreOnlyExt() && StoreOnlyTagCheckingEnabled(accdesc.el) then
        // If the compare on a CAS fails, then it is CONSTRAINED UNPREDICTABLE whether the
        // Tag check is performed.
        if accdesc.tagchecked && cmpfail then
            accdesc.tagchecked = ConstrainUnpredictableBool(Unpredictable_STOREONLYTAGCHECKEDCAS);
```

```
    if HaveMTE2Ext() && accdesc.tagchecked then
        bits(4) ptag = AArch64.PhysicalTag(address);
        if !AArch64.CheckTag(memaddrdesc, accdesc, ptag) then
            accdesc.read = FALSE; // Tag Check Fault on a write.
            AArch64.TagCheckFault(address, accdesc);

if !cmpfail then
    if BigEndian(accdesc.acctype) then
        newvalue = BigEndianReverse(newvalue);
    memstatus = PhysMemWrite(memaddrdesc, bytes, accdesc, newvalue);
    if IsFault(memstatus) then
        HandleExternalWriteAbort(memstatus, memaddrdesc, bytes, accdesc);

if SPESampleInFlight then
    boolean is_load = FALSE;
    SPESampleLoadStore(is_load, accdesc, memaddrdesc);

// Load operations return the old (pre-operation) value
return oldvalue;
```

Library pseudocode for aarch64/functions/memory/MemAtomicRCW

```
// MemAtomicRCW()
// ==============
// Perform a single-copy-atomic access with Read-Check-Write operation

(bits(4), bits(size)) MemAtomicRCW(bits(64) address, bits(size) cmpoperand, bits(size) operand,
                                   AccessDescriptor accdesc_in)
    assert accdesc_in.atomicop;
    assert accdesc_in.rcw;

    constant integer bytes = size DIV 8;
    assert bytes IN {8, 16};

    bits(4) nzcv;
    bits(size) oldvalue;
    bits(size) newvalue;
    AccessDescriptor accdesc = accdesc_in;
    boolean aligned = IsAligned(address, bytes);

    // If the instruction encoding permits tag checking, confer with system register configuration
    // which may override this.
    if HaveMTE2Ext() && accdesc.tagchecked then
        accdesc.tagchecked = AArch64.AccessIsTagChecked(address, accdesc);

    if !aligned && AArch64.UnalignedAccessFaults(accdesc, address, bytes) then
        AArch64.Abort(address, AlignmentFault(accdesc));

    // MMU or MPU lookup
    AddressDescriptor memaddrdesc = AArch64.TranslateAddress(address, accdesc, aligned, size);

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        AArch64.Abort(address, memaddrdesc.fault);

    // Effect on exclusives
    if memaddrdesc.memattrs.shareability != Shareability_NSH then
        ClearExclusiveByAddress(memaddrdesc.paddress, ProcessorID(), size);

    // For Store-only Tag checking, the tag check is performed on the store.
    if (HaveMTE2Ext() && accdesc.tagchecked &&
        (!HaveMTEStoreOnlyExt() || !StoreOnlyTagCheckingEnabled(accdesc.el))) then
        bits(4) ptag = AArch64.PhysicalTag(address);
        if !AArch64.CheckTag(memaddrdesc, accdesc, ptag) then
            AArch64.TagCheckFault(address, accdesc);

    // All observers in the shareability domain observe the following load and store atomically.
    PhysMemRetStatus memstatus;
    (memstatus, oldvalue) = PhysMemRead(memaddrdesc, bytes, accdesc);

    if IsFault(memstatus) then
        HandleExternalReadAbort(memstatus, memaddrdesc, bytes, accdesc);

    if BigEndian(accdesc.acctype) then
        oldvalue = BigEndianReverse(oldvalue);

    boolean cmpfail = FALSE;
    case accdesc.modop of
        when MemAtomicOp_BIC newvalue = oldvalue AND NOT(operand);
        when MemAtomicOp_ORR newvalue = oldvalue OR operand;
        when MemAtomicOp_SWP newvalue = operand;
        when MemAtomicOp_CAS newvalue = operand; cmpfail = oldvalue != cmpoperand;

    if cmpfail then
        nzcv = '1010'; // N = 1 indicates compare failure
    else
        nzcv = RCWCheck(oldvalue, newvalue, accdesc.rcws);

    if HaveMTEStoreOnlyExt() && StoreOnlyTagCheckingEnabled(accdesc.el) then
        // If the compare on a CAS fails, then it is CONSTRAINED UNPREDICTABLE whether the
        // Tag check is performed.
        if accdesc.tagchecked && cmpfail then
```

```
            accdesc.tagchecked = ConstrainUnpredictableBool(Unpredictable_STOREONLYTAGCHECKEDCAS);

        if HaveMTE2Ext() && accdesc.tagchecked then
            bits(4) ptag = AArch64.PhysicalTag(address);
            if !AArch64.CheckTag(memaddrdesc, accdesc, ptag) then
                accdesc.read = FALSE; // Tag Check Fault on a write.
                AArch64.TagCheckFault(address, accdesc);

    if nzcv == '0010' then
        if BigEndian(accdesc.acctype) then
            newvalue = BigEndianReverse(newvalue);

        memstatus = PhysMemWrite(memaddrdesc, bytes, accdesc, newvalue);

        if IsFault(memstatus) then
            HandleExternalWriteAbort(memstatus, memaddrdesc, bytes, accdesc);

    return (nzcv, oldvalue);
```

Library pseudocode for aarch64/functions/memory/MemLoad64B

```
// MemLoad64B()
// ============
// Performs an atomic 64-byte read from a given virtual address.

bits(512) MemLoad64B(bits(64) address, AccessDescriptor accdesc_in)
    bits(512) data;
    constant integer size = 64;
    AccessDescriptor accdesc = accdesc_in;
    boolean aligned = IsAligned(address, size);

    if !aligned && AArch64.UnalignedAccessFaults(accdesc, address, size) then
        AArch64.Abort(address, AlignmentFault(accdesc));

    // If the instruction encoding permits tag checking, confer with system register configuration
    // which may override this.
    if HaveMTE2Ext() && accdesc.tagchecked then
        accdesc.tagchecked = AArch64.AccessIsTagChecked(address, accdesc);

    AddressDescriptor memaddrdesc = AArch64.TranslateAddress(address, accdesc, aligned, size);

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        AArch64.Abort(address, memaddrdesc.fault);

    // Effect on exclusives
    if memaddrdesc.memattrs.shareability != Shareability_NSH then
        ClearExclusiveByAddress(memaddrdesc.paddress, ProcessorID(), size);

    if HaveMTE2Ext() && accdesc.tagchecked then
        bits(4) ptag = AArch64.PhysicalTag(address);
        if !AArch64.CheckTag(memaddrdesc, accdesc, ptag) then
            AArch64.TagCheckFault(address, accdesc);

    if !AddressSupportsLS64(memaddrdesc.paddress.address) then
        c = ConstrainUnpredictable(Unpredictable_LS64UNSUPPORTED);
        assert c IN {Constraint_LIMITED_ATOMICITY, Constraint_FAULT};

        if c == Constraint_FAULT then
            // Generate a stage 1 Data Abort reported using the DFSC code of 110101.
            AArch64.Abort(address, ExclusiveFault(accdesc));
        else
            // Accesses are not single-copy atomic above the byte level.
            for i = 0 to size-1
                PhysMemRetStatus memstatus;
                (memstatus, data<8*i+7:8*i>) = PhysMemRead(memaddrdesc, 1, accdesc);
                if IsFault(memstatus) then
                    HandleExternalReadAbort(memstatus, memaddrdesc, 1, accdesc);

                memaddrdesc.paddress.address = memaddrdesc.paddress.address + 1;
    else
        PhysMemRetStatus memstatus;
        (memstatus, data) = PhysMemRead(memaddrdesc, size, accdesc);
        if IsFault(memstatus) then
            HandleExternalReadAbort(memstatus, memaddrdesc, size, accdesc);

    return data;
```

Library pseudocode for aarch64/functions/memory/MemStore64B

```
// MemStore64B()
// =============
// Performs an atomic 64-byte store to a given virtual address. Function does
// not return the status of the store.

MemStore64B(bits(64) address, bits(512) value, AccessDescriptor accdesc_in)
    constant integer size = 64;
    AccessDescriptor accdesc = accdesc_in;
    boolean aligned = IsAligned(address, size);

    if !aligned && AArch64.UnalignedAccessFaults(accdesc, address, size) then
        AArch64.Abort(address, AlignmentFault(accdesc));

    // If the instruction encoding permits tag checking, confer with system register configuration
    // which may override this.
    if HaveMTE2Ext() && accdesc.tagchecked then
        accdesc.tagchecked = AArch64.AccessIsTagChecked(address, accdesc);

    AddressDescriptor memaddrdesc = AArch64.TranslateAddress(address, accdesc, aligned, size);

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        AArch64.Abort(address, memaddrdesc.fault);

    // Effect on exclusives
    if memaddrdesc.memattrs.shareability != Shareability_NSH then
        ClearExclusiveByAddress(memaddrdesc.paddress, ProcessorID(), 64);

    if HaveMTE2Ext() && accdesc.tagchecked then
        bits(4) ptag = AArch64.PhysicalTag(address);
        if !AArch64.CheckTag(memaddrdesc, accdesc, ptag) then
            AArch64.TagCheckFault(address, accdesc);

    PhysMemRetStatus memstatus;
    if !AddressSupportsLS64(memaddrdesc.paddress.address) then
        c = ConstrainUnpredictable(Unpredictable_LS64UNSUPPORTED);
        assert c IN {Constraint_LIMITED_ATOMICITY, Constraint_FAULT};

        if c == Constraint_FAULT then
            // Generate a Data Abort reported using the DFSC code of 110101.
            AArch64.Abort(address, ExclusiveFault(accdesc));
        else
            // Accesses are not single-copy atomic above the byte level.
            for i = 0 to size-1
                memstatus = PhysMemWrite(memaddrdesc, 1, accdesc, value<8*i+7:8*i>);
                if IsFault(memstatus) then
                    HandleExternalWriteAbort(memstatus, memaddrdesc, 1, accdesc);

                memaddrdesc.paddress.address = memaddrdesc.paddress.address+1;
    else
        memstatus = PhysMemWrite(memaddrdesc, size, accdesc, value);
        if IsFault(memstatus) then
            HandleExternalWriteAbort(memstatus, memaddrdesc, size, accdesc);

    return;
```

Library pseudocode for aarch64/functions/memory/MemStore64BWithRet

```
// MemStore64BWithRet()
// ====================
// Performs an atomic 64-byte store to a given virtual address returning
// the status value of the operation.

bits(64) MemStore64BWithRet(bits(64) address, bits(512) value, AccessDescriptor accdesc_in)
    constant integer size = 64;
    AccessDescriptor accdesc = accdesc_in;
    boolean aligned = IsAligned(address, size);

    if !aligned && AArch64.UnalignedAccessFaults(accdesc, address, size) then
        AArch64.Abort(address, AlignmentFault(accdesc));

    // If the instruction encoding permits tag checking, confer with system register configuration
    // which may override this.
    if HaveMTE2Ext() && accdesc.tagchecked then
        accdesc.tagchecked = AArch64.AccessIsTagChecked(address, accdesc);

    AddressDescriptor memaddrdesc = AArch64.TranslateAddress(address, accdesc, aligned, size);

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        AArch64.Abort(address, memaddrdesc.fault);
        return ZeroExtend('1', 64);

    // Effect on exclusives
    if memaddrdesc.memattrs.shareability != Shareability_NSH then
        ClearExclusiveByAddress(memaddrdesc.paddress, ProcessorID(), 64);

    if HaveMTE2Ext() && accdesc.tagchecked then
        bits(4) ptag = AArch64.PhysicalTag(address);
        if !AArch64.CheckTag(memaddrdesc, accdesc, ptag) then
            AArch64.TagCheckFault(address, accdesc);
            return ZeroExtend('1', 64);

    PhysMemRetStatus memstatus;
    memstatus = PhysMemWrite(memaddrdesc, size, accdesc, value);
    if IsFault(memstatus) then
        HandleExternalWriteAbort(memstatus, memaddrdesc, size, accdesc);

    return memstatus.store64bstatus;
```

Library pseudocode for aarch64/functions/memory/MemStore64BWithRetStatus

```
// MemStore64BWithRetStatus()
// ==========================
// Generates the return status of memory write with ST64BV or ST64BV0
// instructions. The status indicates if the operation succeeded, failed,
// or was not supported at this memory location.

bits(64) MemStore64BWithRetStatus();
```

Library pseudocode for aarch64/functions/memory/NVMem

```
// NVMem[] - non-assignment form
// =============================
// This function is the load memory access for the transformed System register read access
// when Enhanced Nested Virtualization is enabled with HCR_EL2.NV2 = 1.
// The address for the load memory access is calculated using
// the formula SignExtend(VNCR_EL2.BADDR : Offset<11:0>, 64) where,
//  * VNCR_EL2.BADDR holds the base address of the memory location, and
//  * Offset is the unique offset value defined architecturally for each System register that
//    supports transformation of register access to memory access.

bits(64) NVMem[integer offset]
    assert offset > 0;
    constant integer size = 64;
    return NVMem[offset, size];

bits(N) NVMem[integer offset, integer N]
    assert offset > 0;
    assert N IN {64,128};
    bits(64) address = SignExtend(VNCR_EL2.BADDR:offset<11:0>, 64);
    AccessDescriptor accdesc = CreateAccDescNV2(MemOp_LOAD);
    return Mem[address, N DIV 8, accdesc];

// NVMem[] - assignment form
// =========================
// This function is the store memory access for the transformed System register write access
// when Enhanced Nested Virtualization is enabled with HCR_EL2.NV2 = 1.
// The address for the store memory access is calculated using
// the formula SignExtend(VNCR_EL2.BADDR : Offset<11:0>, 64) where,
//  * VNCR_EL2.BADDR holds the base address of the memory location, and
//  * Offset is the unique offset value defined architecturally for each System register that
//    supports transformation of register access to memory access.

NVMem[integer offset] = bits(64) value
    assert offset > 0;
    constant integer size = 64;
    NVMem[offset, size] = value;
    return;

NVMem[integer offset, integer N] = bits(N) value
    assert offset > 0;
    assert N IN {64,128};
    bits(64) address = SignExtend(VNCR_EL2.BADDR:offset<11:0>, 64);
    AccessDescriptor accdesc = CreateAccDescNV2(MemOp_STORE);
    Mem[address, N DIV 8, accdesc] = value;
    return;
```

Library pseudocode for aarch64/functions/memory/PhysMemTagRead

```
// PhysMemTagRead()
// ================
// This is the hardware operation which perform a single-copy atomic,
// Allocation Tag granule aligned, memory access from the tag in PA space.
//
// The function address the array using desc.paddress which supplies:
// * A 52-bit physical address
// * A single NS bit to select between Secure and Non-secure parts of the array.
//
// The accdesc descriptor describes the access type: normal, exclusive, ordered, streaming,
// etc and other parameters required to access the physical memory or for setting syndrome
// register in the event of an External abort.

(PhysMemRetStatus, bits(4)) PhysMemTagRead(AddressDescriptor desc, AccessDescriptor accdesc);
```

Library pseudocode for aarch64/functions/memory/PhysMemTagWrite

```
// PhysMemTagWrite()
// =================
// This is the hardware operation which perform a single-copy atomic,
// Allocation Tag granule aligned, memory access to the tag in PA space.
//
// The function address the array using desc.paddress which supplies:
// * A 52-bit physical address
// * A single NS bit to select between Secure and Non-secure parts of the array.
//
// The accdesc descriptor describes the access type: normal, exclusive, ordered, streaming,
// etc and other parameters required to access the physical memory or for setting syndrome
// register in the event of an External abort.

PhysMemRetStatus PhysMemTagWrite(AddressDescriptor desc, AccessDescriptor accdesc, bits (4) value);
```

Library pseudocode for aarch64/functions/memory/StoreOnlyTagCheckingEnabled

```
// StoreOnlyTagCheckingEnabled()
// =============================
// Returns TRUE if loads executed at the given Exception level are Tag unchecked.

boolean StoreOnlyTagCheckingEnabled(bits(2) el)
    assert HaveMTEStoreOnlyExt();
    bit tcso;

    case el of
        when EL0
            if !ELIsInHost(el) then
                tcso = SCTLR_EL1.TCSO0;
            else
                tcso = SCTLR_EL2.TCSO0;
        when EL1
            tcso = SCTLR_EL1.TCSO;
        when EL2
            tcso = SCTLR_EL2.TCSO;
        otherwise
            tcso = SCTLR_EL3.TCSO;

    return tcso == '1';
```

Library pseudocode for aarch64/functions/mops/CPYFOptionA

```
// CPYFOptionA()
// =============
// Returns TRUE if the implementation uses Option A for the
// CPYF* instructions, and FALSE otherwise.

boolean CPYFOptionA()
    return boolean IMPLEMENTATION_DEFINED "CPYF* instructions use Option A";
```

Library pseudocode for aarch64/functions/mops/CPYOptionA

```
// CPYOptionA()
// ============
// Returns TRUE if the implementation uses Option A for the
// CPY* instructions, and FALSE otherwise.

boolean CPYOptionA()
    return boolean IMPLEMENTATION_DEFINED "CPY* instructions use Option A";
```

Library pseudocode for aarch64/functions/mops/CPYPostSizeChoice

```
// CPYPostSizeChoice()
// ===================
// Returns the size of the copy that is performed by the CPYE* instructions for this
// implementation given the parameters of the destination, source and size of the copy.
// Postsize is encoded as -1*size for an option A implementation if cpysize is negative.

bits(64) CPYPostSizeChoice(bits(64) toaddress, bits(64) fromaddress, bits(64) cpysize);
```

Library pseudocode for aarch64/functions/mops/CPYPreSizeChoice

```
// CPYPreSizeChoice()
// ==================
// Returns the size of the copy that is performed by the CPYP* instructions for this
// implementation given the parameters of the destination, source and size of the copy.
// Presize is encoded as -1*size for an option A implementation if cpysize is negative.

bits(64) CPYPreSizeChoice(bits(64) toaddress, bits(64) fromaddress, bits(64) cpysize);
```

Library pseudocode for aarch64/functions/mops/CPYSizeChoice

```
// CPYSizeChoice()
// ===============
// Returns the size of the block this performed for an iteration of the copy given the
// parameters of the destination, source and size of the copy.

integer CPYSizeChoice(bits(64) toaddress, bits(64) fromaddress, bits(64) cpysize);
```

Library pseudocode for aarch64/functions/mops/CheckMOPSEnabled

```
// CheckMOPSEnabled()
// ==================
// Check for EL0 and EL1 access to the CPY* and SET* instructions.

CheckMOPSEnabled()
    if (PSTATE.EL IN {EL0, EL1} && EL2Enabled() &&
        (HCR_EL2.TGE == '0' || HCR_EL2.E2H == '0') &&
        (!IsHCRXEL2Enabled() || HCRX_EL2.MSCEn == '0')) then
        UNDEFINED;

    if (PSTATE.EL == EL0 && SCTLR_EL1.MSCEn == '0' &&
        (!EL2Enabled() || HCR_EL2.TGE == '0' || HCR_EL2.E2H == '0')) then
        UNDEFINED;

    if PSTATE.EL == EL0 && IsInHost() && SCTLR_EL2.MSCEn == '0' then
        UNDEFINED;
```

Library pseudocode for aarch64/functions/mops/MOPSStage

```
// MOPSStage
// =========

enumeration MOPSStage { MOPSStage_Prologue, MOPSStage_Main, MOPSStage_Epilogue };
```

Library pseudocode for aarch64/functions/mops/MaxBlockSizeCopiedBytes

```
// MaxBlockSizeCopiedBytes()
// =========================
// Returns the maximum number of bytes that can used in a single block of the copy.

integer MaxBlockSizeCopiedBytes()
    return integer IMPLEMENTATION_DEFINED "Maximum bytes used in a single block of a copy";
```

Library pseudocode for aarch64/functions/mops/MemCpyDirectionChoice

```
// MemCpyDirectionChoice()
// =======================
// Returns true if in the non-overlapping case of a memcpy of size cpysize bytes
// from the source address fromaddress to destination address toaddress is done
// in the forward direction on this implementation.

boolean MemCpyDirectionChoice(bits(64) fromaddress, bits(64) toaddress, bits(64) cpysize);
```

Library pseudocode for aarch64/functions/mops/MemCpyParametersIllformedE

```
// MemCpyParametersIllformedE()
// ============================
// Returns TRUE if the inputs are not well formed (in terms of their size and/or alignment)
// for a CPYE* instruction for this implementation given the parameters of the destination,
// source and size of the copy.

boolean MemCpyParametersIllformedE(bits(64) toaddress, bits(64) fromaddress,
                                   bits(64) cpysize);
```

Library pseudocode for aarch64/functions/mops/MemCpyParametersIllformedM

```
// MemCpyParametersIllformedM()
// ============================
// Returns TRUE if the inputs are not well formed (in terms of their size and/or alignment)
// for a CPYM* instruction for this implementation given the parameters of the destination,
// source and size of the copy.

boolean MemCpyParametersIllformedM(bits(64) toaddress, bits(64) fromaddress,
                                   bits(64) cpysize);
```

Library pseudocode for aarch64/functions/mops/MemCpyZeroSizeCheck

```
// MemCpyZeroSizeCheck()
// =====================
// Returns TRUE if the implementation option is checked on a copy of size zero remaining.

boolean MemCpyZeroSizeCheck();
```

Library pseudocode for aarch64/functions/mops/MemSetParametersIllformedE

```
// MemSetParametersIllformedE()
// ============================
// Returns TRUE if the inputs are not well formed (in terms of their size and/or
// alignment) for a SETE* or SETGE* instruction for this implementation given the
// parameters of the destination and size of the set.

boolean MemSetParametersIllformedE(bits(64) toaddress, bits(64) setsize,
                                   boolean IsSETGE);
```

Library pseudocode for aarch64/functions/mops/MemSetParametersIllformedM

```
// MemSetParametersIllformedM()
// ============================
// Returns TRUE if the inputs are not well formed (in terms of their size and/or
// alignment) for a SETM* or SETGM* instruction for this implementation given the
// parameters of the destination and size of the copy.

boolean MemSetParametersIllformedM(bits(64) toaddress, bits(64) setsize,
                                   boolean IsSETGM);
```

Library pseudocode for aarch64/functions/mops/MemSetZeroSizeCheck

```
// MemSetZeroSizeCheck()
// =====================
// Returns TRUE if the implementation option is checked on a copy of size zero remaining.

boolean MemSetZeroSizeCheck();
```

Library pseudocode for aarch64/functions/mops/MismatchedCpySetTargetEL

```
// MismatchedCpySetTargetEL()
// ==========================
// Return the target exception level for an Exception_MemCpyMemSet.

bits(2) MismatchedCpySetTargetEL()
    bits(2) target_el;

    if UInt(PSTATE.EL) > UInt(EL1) then
        target_el = PSTATE.EL;
    elsif PSTATE.EL == EL0 && EL2Enabled() && HCR_EL2.TGE == '1' then
        target_el = EL2;
    elsif (PSTATE.EL == EL1 && EL2Enabled() &&
        IsHCRXEL2Enabled() && HCRX_EL2.MCE2 == '1') then
        target_el = EL2;
    else
        target_el = EL1;

    return target_el;
```

Library pseudocode for aarch64/functions/mops/MismatchedMemCpyException

```
// MismatchedMemCpyException()
// ===========================
// Generates an exception for a CPY* instruction if the version
// is inconsistent with the state of the call.

MismatchedMemCpyException(boolean option_a, integer destreg, integer srcreg, integer sizereg,
                          boolean wrong_option, boolean from_epilogue, bits(4) options)
    bits(64) preferred_exception_return = ThisInstrAddr(64);
    integer vect_offset = 0x0;
    bits(2) target_el = MismatchedCpySetTargetEL();

    ExceptionRecord except = ExceptionSyndrome(Exception_MemCpyMemSet);
    except.syndrome<24>    = '0';
    except.syndrome<23>    = '0';
    except.syndrome<22:19> = options;
    except.syndrome<18>    = if from_epilogue then '1' else '0';
    except.syndrome<17>    = if wrong_option then '1' else '0';
    except.syndrome<16>    = if option_a then '1' else '0';
    // exception.syndrome<15> is RES0
    except.syndrome<14:10> = destreg<4:0>;
    except.syndrome<9:5>   = srcreg<4:0>;
    except.syndrome<4:0>   = sizereg<4:0>;

    AArch64.TakeException(target_el, except, preferred_exception_return, vect_offset);
```

Library pseudocode for aarch64/functions/mops/MismatchedMemSetException

```
// MismatchedMemSetException()
// ===========================
// Generates an exception for a SET* instruction if the version
// is inconsistent with the state of the call.

MismatchedMemSetException(boolean option_a, integer destreg, integer datareg, integer sizereg,
                          boolean wrong_option, boolean from_epilogue, bits(2) options,
                          boolean is_SETG)
    bits(64) preferred_exception_return = ThisInstrAddr(64);
    integer vect_offset = 0x0;
    bits(2) target_el = MismatchedCpySetTargetEL();

    ExceptionRecord except = ExceptionSyndrome(Exception_MemCpyMemSet);
    except.syndrome<24>    = '1';
    except.syndrome<23>    = if is_SETG then '1' else '0';
    // exception.syndrome<22:21> is RES0
    except.syndrome<20:19> = options;
    except.syndrome<18>    = if from_epilogue then '1' else '0';
    except.syndrome<17>    = if wrong_option then '1' else '0';
    except.syndrome<16>    = if option_a then '1' else '0';
    // exception.syndrome<15> is RES0
    except.syndrome<14:10> = destreg<4:0>;
    except.syndrome<9:5>   = datareg<4:0>;
    except.syndrome<4:0>   = sizereg<4:0>;

    AArch64.TakeException(target_el, except, preferred_exception_return, vect_offset);
```

Library pseudocode for aarch64/functions/mops/SETGOptionA

```
// SETGOptionA()
// =============
// Returns TRUE if the implementation uses Option A for the
// SETG* instructions, and FALSE otherwise.

boolean SETGOptionA()
    return boolean IMPLEMENTATION_DEFINED "SETG* instructions use Option A";
```

Library pseudocode for aarch64/functions/mops/SETOptionA

```
// SETOptionA()
// ============
// Returns TRUE if the implementation uses Option A for the
// SET* instructions, and FALSE otherwise.

boolean SETOptionA()
    return boolean IMPLEMENTATION_DEFINED "SET* instructions use Option A";
```

Library pseudocode for aarch64/functions/mops/SETPostSizeChoice

```
// SETPostSizeChoice()
// ===================
// Returns the size of the set that is performed by the SETE* or SETGE* instructions
// for this implementation, given the parameters of the destination and size of the set.
// Postsize is encoded as -1*size for an option A implementation if setsize is negative.

bits(64) SETPostSizeChoice(bits(64) toaddress, bits(64) setsize, boolean IsSETGE);
```

Library pseudocode for aarch64/functions/mops/SETPreSizeChoice

```
// SETPreSizeChoice()
// ==================
// Returns the size of the set that is performed by the SETP* or SETGP* instructions
// for this implementation, given the parameters of the destination and size of the set.
// Presize is encoded as -1*size for an option A implementation if setsize is negative.

bits(64) SETPreSizeChoice(bits(64) toaddress, bits(64) setsize, boolean IsSETGP);
```

Library pseudocode for aarch64/functions/mops/SETSizeChoice

```
// SETSizeChoice()
// ===============
// Returns the size of the block thisperformed for an iteration of the set given
// the parameters of the destination and size of the set. The size of the block
// is an integer multiple of AlignSize.

integer SETSizeChoice(bits(64) toaddress, bits(64) setsize, integer AlignSize);
```

Library pseudocode for aarch64/functions/movewideop/MoveWideOp

```
// MoveWideOp
// ==========
// Move wide 16-bit immediate instruction types.

enumeration MoveWideOp  {MoveWideOp_N, MoveWideOp_Z, MoveWideOp_K};
```

Library pseudocode for aarch64/functions/movwpreferred/MoveWidePreferred

```
// MoveWidePreferred()
// ===================
//
// Return TRUE if a bitmask immediate encoding would generate an immediate
// value that could also be represented by a single MOVZ or MOVN instruction.
// Used as a condition for the preferred MOV<-ORR alias.

boolean MoveWidePreferred(bit sf, bit immN, bits(6) imms, bits(6) immr)
    integer s = UInt(imms);
    integer r = UInt(immr);
    integer width = if sf == '1' then 64 else 32;

    // element size must equal total immediate size
    if sf == '1' && !((immN:imms) IN {'1xxxxxx'}) then
        return FALSE;
    if sf == '0' && !((immN:imms) IN {'00xxxxx'}) then
        return FALSE;

    // for MOVZ must contain no more than 16 ones
    if s < 16 then
        // ones must not span halfword boundary when rotated
        return (-r MOD 16) <= (15 - s);

    // for MOVN must contain no more than 16 zeros
    if s >= width - 15 then
        // zeros must not span halfword boundary when rotated
        return (r MOD 16) <= (s - (width - 15));

    return FALSE;
```

Library pseudocode for aarch64/functions/pac/addpac/AddPAC

```
// AddPAC()
// ========
// Calculates the pointer authentication code for a 64-bit quantity and then
// inserts that into pointer authentication code field of that 64-bit quantity.

bits(64) AddPAC(bits(64) ptr, bits(64) modifier, bits(128) K, boolean data)
    bits(64) PAC;
    bits(64) result;
    bits(64) ext_ptr;
    bits(64) extfield;
    bit selbit;
    boolean isgeneric = FALSE;
    boolean tbi = EffectiveTBI(ptr, !data, PSTATE.EL) == '1';
    boolean mtx = EffectiveMTX(ptr, !data, PSTATE.EL) == '1';
    integer top_bit = if tbi then 55 else 63;

    // If tagged pointers are in use for a regime with two TTBRs, use bit<55> of
    // the pointer to select between upper and lower ranges, and preserve this.
    // This handles the awkward case where there is apparently no correct choice between
    // the upper and lower address range - ie an addr of 1xxxxxxx0... with TBI0=0 and TBI1=1
    // and 0xxxxxxx1 with TBI1=0 and TBI0=1:
    if PtrHasUpperAndLowerAddRanges() then
        assert S1TranslationRegime() IN {EL1, EL2};
        if S1TranslationRegime() == EL1 then
            // EL1 translation regime registers
            if data then
                if TCR_EL1.TBI1 == '1' || TCR_EL1.TBI0 == '1' then
                    selbit = ptr<55>;
                else
                    selbit = ptr<63>;
            else
                if ((TCR_EL1.TBI1 == '1' && TCR_EL1.TBID1 == '0') ||
                    (TCR_EL1.TBI0 == '1' && TCR_EL1.TBID0 == '0')) then
                    selbit = ptr<55>;
                else
                    selbit = ptr<63>;
        else
            // EL2 translation regime registers
            if data then
                if TCR_EL2.TBI1 == '1' || TCR_EL2.TBI0 == '1' then
                    selbit = ptr<55>;
                else
                    selbit = ptr<63>;
            else
                if ((TCR_EL2.TBI1 == '1' && TCR_EL2.TBID1 == '0') ||
                    (TCR_EL2.TBI0 == '1' && TCR_EL2.TBID0 == '0')) then
                    selbit = ptr<55>;
                else
                    selbit = ptr<63>;
    else selbit = if tbi then ptr<55> else ptr<63>;

    if HaveEnhancedPAC2() && ConstPACField() then selbit = ptr<55>;
    integer bottom_PAC_bit = CalculateBottomPACBit(selbit);

    // If the VA is 56 or 55 bits and Top Byte is Ignored,
    // there are no unused bits left to insert the PAC
    if tbi && bottom_PAC_bit >= 55 then
        return ptr;

    extfield = Replicate(selbit, 64);

    // Compute the pointer authentication code for a ptr with good extension bits
    if tbi then
        ext_ptr = (ptr<63:56> :
                   extfield<(56-bottom_PAC_bit)-1:0> : ptr<bottom_PAC_bit-1:0>);
    elsif mtx then
        ext_ptr = (extfield<63:60> : ptr<59:56> :
                   extfield<(56-bottom_PAC_bit)-1:0> : ptr<bottom_PAC_bit-1:0>);
    else
        ext_ptr =  extfield<(64-bottom_PAC_bit)-1:0> : ptr<bottom_PAC_bit-1:0>;
```

```
PAC = ComputePAC(ext_ptr, modifier, K<127:64>, K<63:0>, isgeneric);

// Check if the ptr has good extension bits and corrupt the pointer authentication code if not
bits(64) unusedbits_mask = Zeros(64);
unusedbits_mask<54:bottom_PAC_bit> = Ones((54-bottom_PAC_bit)+1);
if tbi then
    unusedbits_mask<63:56> = Ones(8);
elsif mtx then
    unusedbits_mask<63:60> = Ones(4);
if !IsZero(ptr AND unusedbits_mask) && ((ptr AND unusedbits_mask) != unusedbits_mask) then
    if HaveEnhancedPAC() then
        PAC = 0x0000000000000000<63:0>;
    elsif !HaveEnhancedPAC2() then
        PAC<top_bit-1> = NOT(PAC<top_bit-1>);

// Preserve the determination between upper and lower address at bit<55> and insert PAC into
// bits that are not used for the address or the tag(s).
if !HaveEnhancedPAC2() then
    if tbi then
        result = ptr<63:56>:selbit:PAC<54:bottom_PAC_bit>:ptr<bottom_PAC_bit-1:0>;
    else
        result = PAC<63:56>:selbit:PAC<54:bottom_PAC_bit>:ptr<bottom_PAC_bit-1:0>;
        // A compliant implementation of FEAT_MTE4 also implements FEAT_PAuth2
        assert !mtx;
else
    if tbi then
        result = (ptr<63:56>                               : selbit :
                  (ptr<54:bottom_PAC_bit> EOR PAC<54:bottom_PAC_bit>) :
                  ptr<bottom_PAC_bit-1:0>);
    elsif mtx then
        result = ((ptr<63:60> EOR PAC<63:60>) : ptr<59:56> : selbit :
                  (ptr<54:bottom_PAC_bit> EOR PAC<54:bottom_PAC_bit>) :
                  ptr<bottom_PAC_bit-1:0>);
    else
        result = ((ptr<63:56> EOR PAC<63:56>)              : selbit :
                  (ptr<54:bottom_PAC_bit> EOR PAC<54:bottom_PAC_bit>) :
                  ptr<bottom_PAC_bit-1:0>);
return result;
```

Library pseudocode for aarch64/functions/pac/addpacda/AddPACDA

```
// AddPACDA()
// ==========
// Returns a 64-bit value containing x, but replacing the pointer authentication code
// field bits with a pointer authentication code, where the pointer authentication
// code is derived using a cryptographic algorithm as a combination of x, y and the
// APDAKey_EL1.

bits(64) AddPACDA(bits(64) x, bits(64) y)
    boolean TrapEL2;
    boolean TrapEL3;
    bits(1)  Enable;
    bits(128) APDAKey_EL1;

    APDAKey_EL1 = APDAKeyHi_EL1<63:0> : APDAKeyLo_EL1<63:0>;
    case PSTATE.EL of
        when EL0
            boolean IsEL1Regime = S1TranslationRegime() == EL1;
            Enable = if IsEL1Regime then SCTLR_EL1.EnDA else SCTLR_EL2.EnDA;
            TrapEL2 = (EL2Enabled() && HCR_EL2.API == '0' &&
                        (HCR_EL2.TGE == '0' || HCR_EL2.E2H == '0'));
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL1
            Enable = SCTLR_EL1.EnDA;
            TrapEL2 = EL2Enabled() && HCR_EL2.API == '0';
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL2
            Enable = SCTLR_EL2.EnDA;
            TrapEL2 = FALSE;
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL3
            Enable = SCTLR_EL3.EnDA;
            TrapEL2 = FALSE;
            TrapEL3 = FALSE;

    if Enable == '0' then
        return x;
    elsif TrapEL3 && EL3SDDUndefPriority() then
        UNDEFINED;
    elsif TrapEL2 then
        TrapPACUse(EL2);
    elsif TrapEL3 then
        if EL3SDDUndef() then
            UNDEFINED;
        else
            TrapPACUse(EL3);
    else
        return AddPAC(x, y, APDAKey_EL1, TRUE);
```

Library pseudocode for aarch64/functions/pac/addpacdb/AddPACDB

```
// AddPACDB()
// ==========
// Returns a 64-bit value containing x, but replacing the pointer authentication code
// field bits with a pointer authentication code, where the pointer authentication
// code is derived using a cryptographic algorithm as a combination of x, y and the
// APDBKey_EL1.

bits(64) AddPACDB(bits(64) x, bits(64) y)
    boolean TrapEL2;
    boolean TrapEL3;
    bits(1)  Enable;
    bits(128) APDBKey_EL1;

    APDBKey_EL1 = APDBKeyHi_EL1<63:0> : APDBKeyLo_EL1<63:0>;
    case PSTATE.EL of
        when EL0
            boolean IsEL1Regime = S1TranslationRegime() == EL1;
            Enable = if IsEL1Regime then SCTLR_EL1.EnDB else SCTLR_EL2.EnDB;
            TrapEL2 = (EL2Enabled() && HCR_EL2.API == '0' &&
                       (HCR_EL2.TGE == '0' || HCR_EL2.E2H == '0'));
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL1
            Enable = SCTLR_EL1.EnDB;
            TrapEL2 = EL2Enabled() && HCR_EL2.API == '0';
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL2
            Enable = SCTLR_EL2.EnDB;
            TrapEL2 = FALSE;
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL3
            Enable = SCTLR_EL3.EnDB;
            TrapEL2 = FALSE;
            TrapEL3 = FALSE;

    if Enable == '0' then
        return x;
    elsif TrapEL3 && EL3SDDUndefPriority() then
        UNDEFINED;
    elsif TrapEL2 then
        TrapPACUse(EL2);
    elsif TrapEL3 then
        if EL3SDDUndef() then
            UNDEFINED;
        else
            TrapPACUse(EL3);
    else
        return AddPAC(x, y, APDBKey_EL1, TRUE);
```

Library pseudocode for aarch64/functions/pac/addpacga/AddPACGA

```
// AddPACGA()
// ==========
// Returns a 64-bit value where the lower 32 bits are 0, and the upper 32 bits contain
// a 32-bit pointer authentication code which is derived using a cryptographic
// algorithm as a combination of x, y and the APGAKey_EL1.

bits(64) AddPACGA(bits(64) x, bits(64) y)
    boolean TrapEL2;
    boolean TrapEL3;
    bits(128) APGAKey_EL1;
    boolean isgeneric = TRUE;

    APGAKey_EL1 = APGAKeyHi_EL1<63:0> : APGAKeyLo_EL1<63:0>;
    case PSTATE.EL of
        when EL0
            TrapEL2 = (EL2Enabled() && HCR_EL2.API == '0' &&
                        (HCR_EL2.TGE == '0' || HCR_EL2.E2H == '0'));
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL1
            TrapEL2 = EL2Enabled() && HCR_EL2.API == '0';
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL2
            TrapEL2 = FALSE;
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL3
            TrapEL2 = FALSE;
            TrapEL3 = FALSE;

    if TrapEL3 && EL3SDDUndefPriority() then
        UNDEFINED;
    elsif TrapEL2 then
        TrapPACUse(EL2);
    elsif TrapEL3 then
        if EL3SDDUndef() then
            UNDEFINED;
        else
            TrapPACUse(EL3);
    else
        return ComputePAC(x, y, APGAKey_EL1<127:64>, APGAKey_EL1<63:0>, isgeneric)<63:32>:Zeros(32);
```

Library pseudocode for aarch64/functions/pac/addpacia/AddPACIA

```
// AddPACIA()
// ==========
// Returns a 64-bit value containing x, but replacing the pointer authentication code
// field bits with a pointer authentication code, where the pointer authentication
// code is derived using a cryptographic algorithm as a combination of x, y, and the
// APIAKey_EL1.

bits(64) AddPACIA(bits(64) x, bits(64) y)
    boolean TrapEL2;
    boolean TrapEL3;
    bits(1)  Enable;
    bits(128) APIAKey_EL1;

    APIAKey_EL1 = APIAKeyHi_EL1<63:0>:APIAKeyLo_EL1<63:0>;
    case PSTATE.EL of
        when EL0
            boolean IsEL1Regime = S1TranslationRegime() == EL1;
            Enable = if IsEL1Regime then SCTLR_EL1.EnIA else SCTLR_EL2.EnIA;
            TrapEL2 = (EL2Enabled() && HCR_EL2.API == '0' &&
                       (HCR_EL2.TGE == '0' || HCR_EL2.E2H == '0'));
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL1
            Enable = SCTLR_EL1.EnIA;
            TrapEL2 = EL2Enabled() && HCR_EL2.API == '0';
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL2
            Enable = SCTLR_EL2.EnIA;
            TrapEL2 = FALSE;
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL3
            Enable = SCTLR_EL3.EnIA;
            TrapEL2 = FALSE;
            TrapEL3 = FALSE;

    if Enable == '0' then
        return x;
    elsif TrapEL3 && EL3SDDUndefPriority() then
        UNDEFINED;
    elsif TrapEL2 then
        TrapPACUse(EL2);
    elsif TrapEL3 then
        if EL3SDDUndef() then
            UNDEFINED;
        else
            TrapPACUse(EL3);
    else
        return AddPAC(x, y, APIAKey_EL1, FALSE);
```

Library pseudocode for aarch64/functions/pac/addpacib/AddPACIB

```
// AddPACIB()
// ==========
// Returns a 64-bit value containing x, but replacing the pointer authentication code
// field bits with a pointer authentication code, where the pointer authentication
// code is derived using a cryptographic algorithm as a combination of x, y and the
// APIBKey_EL1.

bits(64) AddPACIB(bits(64) x, bits(64) y)
    boolean TrapEL2;
    boolean TrapEL3;
    bits(1)  Enable;
    bits(128) APIBKey_EL1;

    APIBKey_EL1 = APIBKeyHi_EL1<63:0> : APIBKeyLo_EL1<63:0>;
    case PSTATE.EL of
        when EL0
            boolean IsEL1Regime = S1TranslationRegime() == EL1;
            Enable = if IsEL1Regime then SCTLR_EL1.EnIB else SCTLR_EL2.EnIB;
            TrapEL2 = (EL2Enabled() && HCR_EL2.API == '0' &&
                       (HCR_EL2.TGE == '0' || HCR_EL2.E2H == '0'));
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL1
            Enable = SCTLR_EL1.EnIB;
            TrapEL2 = EL2Enabled() && HCR_EL2.API == '0';
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL2
            Enable = SCTLR_EL2.EnIB;
            TrapEL2 = FALSE;
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL3
            Enable = SCTLR_EL3.EnIB;
            TrapEL2 = FALSE;
            TrapEL3 = FALSE;

    if Enable == '0' then
        return x;
    elsif TrapEL3 && EL3SDDUndefPriority() then
        UNDEFINED;
    elsif TrapEL2 then
        TrapPACUse(EL2);
    elsif TrapEL3 then
        if EL3SDDUndef() then
            UNDEFINED;
        else
            TrapPACUse(EL3);
    else
        return AddPAC(x, y, APIBKey_EL1, FALSE);
```

Library pseudocode for aarch64/functions/pac/auth/AArch64.PACFailException

```
// AArch64.PACFailException()
// ==========================
// Generates a PAC Fail Exception

AArch64.PACFailException(bits(2) syndrome)
    route_to_el2 = PSTATE.EL == EL0 && EL2Enabled() && HCR_EL2.TGE == '1';
    bits(64) preferred_exception_return = ThisInstrAddr(64);
    vect_offset = 0x0;

    except = ExceptionSyndrome(Exception_PACFail);
    except.syndrome<1:0>   = syndrome;
    except.syndrome<24:2>  = Zeros(23);                // RES0

    if UInt(PSTATE.EL) > UInt(EL0) then
        AArch64.TakeException(PSTATE.EL, except, preferred_exception_return, vect_offset);
    elsif route_to_el2 then
        AArch64.TakeException(EL2, except, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(EL1, except, preferred_exception_return, vect_offset);
```

Library pseudocode for aarch64/functions/pac/auth/Auth

```
// Auth()
// ======
// Restores the upper bits of the address to be all zeros or all ones (based on the
// value of bit[55]) and computes and checks the pointer authentication code. If the
// check passes, then the restored address is returned. If the check fails, the
// second-top and third-top bits of the extension bits in the pointer authentication code
// field are corrupted to ensure that accessing the address will give a translation fault.

bits(64) Auth(bits(64) ptr, bits(64) modifier, bits(128) K, boolean data, bit key_number,
              boolean is_combined)
    bits(64) PAC;
    bits(64) result;
    bits(64) original_ptr;
    bits(2) error_code;
    bits(64) extfield;
    boolean isgeneric = FALSE;

    // Reconstruct the extension field used of adding the PAC to the pointer
    boolean tbi = EffectiveTBI(ptr, !data, PSTATE.EL) == '1';
    boolean mtx = EffectiveMTX(ptr, !data, PSTATE.EL) == '1';
    integer bottom_PAC_bit = CalculateBottomPACBit(ptr<55>);
    extfield = Replicate(ptr<55>, 64);

    // If the VA is 56 or 55 bits and Top Byte is Ignored,
    // there are no unused bits left for the PAC
    if tbi && bottom_PAC_bit >= 55 then
        return ptr;

    if tbi then
        original_ptr = (ptr<63:56> :
                        extfield<(56-bottom_PAC_bit)-1:0> : ptr<bottom_PAC_bit-1:0>);
    elsif mtx then
        original_ptr = (extfield<63:60> : ptr<59:56> :
                        extfield<(56-bottom_PAC_bit)-1:0> : ptr<bottom_PAC_bit-1:0>);
    else
        original_ptr =  extfield<(64-bottom_PAC_bit)-1:0> : ptr<bottom_PAC_bit-1:0>;

    PAC = ComputePAC(original_ptr, modifier, K<127:64>, K<63:0>, isgeneric);
    // Check pointer authentication code
    if tbi then
        if !HaveEnhancedPAC2() then
            if PAC<54:bottom_PAC_bit> == ptr<54:bottom_PAC_bit> then
                result = original_ptr;
            else
                error_code = key_number:NOT(key_number);
                result = original_ptr<63:55>:error_code:original_ptr<52:0>;
        else
            result = ptr;
            result<54:bottom_PAC_bit> = result<54:bottom_PAC_bit> EOR PAC<54:bottom_PAC_bit>;
            if HaveFPACCombined() || (HaveFPAC() && !is_combined) then
                if result<54:bottom_PAC_bit> != Replicate(result<55>, (55-bottom_PAC_bit)) then
                    error_code = (if data then '1' else '0'):key_number;
                    AArch64.PACFailException(error_code);
    elsif mtx then
        assert HaveEnhancedPAC2();
        result = ptr;
        result<54:bottom_PAC_bit> = result<54:bottom_PAC_bit> EOR PAC<54:bottom_PAC_bit>;
        result<63:60> = result<63:60> EOR PAC<63:60>;
        if HaveFPACCombined() || (HaveFPAC() && !is_combined) then
            if ((result<54:bottom_PAC_bit> != Replicate(result<55>, (55-bottom_PAC_bit))) ||
                (result<63:60> != Replicate(result<55>, 4))) then
                error_code = (if data then '1' else '0'):key_number;
                AArch64.PACFailException(error_code);
    else
        if !HaveEnhancedPAC2() then
            if PAC<54:bottom_PAC_bit> == ptr<54:bottom_PAC_bit> && PAC<63:56> == ptr<63:56> then
                result = original_ptr;
            else
                error_code = key_number:NOT(key_number);
                result = original_ptr<63>:error_code:original_ptr<60:0>;
```

```
        else
            result = ptr;
            result<54:bottom_PAC_bit> = result<54:bottom_PAC_bit> EOR PAC<54:bottom_PAC_bit>;
            result<63:56> = result<63:56> EOR PAC<63:56>;
            if HaveFPACCombined() || (HaveFPAC() && !is_combined) then
                if result<63:bottom_PAC_bit> != Replicate(result<55>, (64-bottom_PAC_bit)) then
                    error_code = (if data then '1' else '0'):key_number;
                    AArch64.PACFailException(error_code);
    return result;
```

Library pseudocode for aarch64/functions/pac/authda/AuthDA

```
// AuthDA()
// ========
// Returns a 64-bit value containing x, but replacing the pointer authentication code
// field bits with the extension of the address bits. The instruction checks a pointer
// authentication code in the pointer authentication code field bits of x, using the same
// algorithm and key as AddPACDA().

bits(64) AuthDA(bits(64) x, bits(64) y, boolean is_combined)
    boolean TrapEL2;
    boolean TrapEL3;
    bits(1)  Enable;
    bits(128) APDAKey_EL1;

    APDAKey_EL1 = APDAKeyHi_EL1<63:0> : APDAKeyLo_EL1<63:0>;
    case PSTATE.EL of
        when EL0
            boolean IsEL1Regime = S1TranslationRegime() == EL1;
            Enable = if IsEL1Regime then SCTLR_EL1.EnDA else SCTLR_EL2.EnDA;
            TrapEL2 = (EL2Enabled() && HCR_EL2.API == '0' &&
                      (HCR_EL2.TGE == '0' || HCR_EL2.E2H == '0'));
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL1
            Enable = SCTLR_EL1.EnDA;
            TrapEL2 = EL2Enabled() && HCR_EL2.API == '0';
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL2
            Enable = SCTLR_EL2.EnDA;
            TrapEL2 = FALSE;
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL3
            Enable = SCTLR_EL3.EnDA;
            TrapEL2 = FALSE;
            TrapEL3 = FALSE;

    if Enable == '0' then
        return x;
    elsif TrapEL3 && EL3SDDUndefPriority() then
        UNDEFINED;
    elsif TrapEL2 then
        TrapPACUse(EL2);
    elsif TrapEL3 then
        if EL3SDDUndef() then
            UNDEFINED;
        else
            TrapPACUse(EL3);
    else
        return Auth(x, y, APDAKey_EL1, TRUE, '0', is_combined);
```

Library pseudocode for aarch64/functions/pac/authdb/AuthDB

```
// AuthDB()
// ========
// Returns a 64-bit value containing x, but replacing the pointer authentication code
// field bits with the extension of the address bits. The instruction checks a
// pointer authentication code in the pointer authentication code field bits of x, using
// the same algorithm and key as AddPACDB().

bits(64) AuthDB(bits(64) x, bits(64) y, boolean is_combined)
    boolean TrapEL2;
    boolean TrapEL3;
    bits(1)  Enable;
    bits(128) APDBKey_EL1;

    APDBKey_EL1 = APDBKeyHi_EL1<63:0> : APDBKeyLo_EL1<63:0>;
    case PSTATE.EL of
        when EL0
            boolean IsEL1Regime = S1TranslationRegime() == EL1;
            Enable = if IsEL1Regime then SCTLR_EL1.EnDB else SCTLR_EL2.EnDB;
            TrapEL2 = (EL2Enabled() && HCR_EL2.API == '0' &&
                        (HCR_EL2.TGE == '0' || HCR_EL2.E2H == '0'));
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL1
            Enable = SCTLR_EL1.EnDB;
            TrapEL2 = EL2Enabled() && HCR_EL2.API == '0';
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL2
            Enable = SCTLR_EL2.EnDB;
            TrapEL2 = FALSE;
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL3
            Enable = SCTLR_EL3.EnDB;
            TrapEL2 = FALSE;
            TrapEL3 = FALSE;

    if Enable == '0' then
        return x;
    elsif TrapEL3 && EL3SDDUndefPriority() then
        UNDEFINED;
    elsif TrapEL2 then
        TrapPACUse(EL2);
    elsif TrapEL3 then
        if EL3SDDUndef() then
            UNDEFINED;
        else
            TrapPACUse(EL3);
    else
        return Auth(x, y, APDBKey_EL1, TRUE, '1', is_combined);
```

Library pseudocode for aarch64/functions/pac/authia/AuthIA

```
// AuthIA()
// ========
// Returns a 64-bit value containing x, but replacing the pointer authentication code
// field bits with the extension of the address bits. The instruction checks a pointer
// authentication code in the pointer authentication code field bits of x, using the same
// algorithm and key as AddPACIA().

bits(64) AuthIA(bits(64) x, bits(64) y, boolean is_combined)
    boolean TrapEL2;
    boolean TrapEL3;
    bits(1)  Enable;
    bits(128) APIAKey_EL1;

    APIAKey_EL1 = APIAKeyHi_EL1<63:0> : APIAKeyLo_EL1<63:0>;
    case PSTATE.EL of
        when EL0
            boolean IsEL1Regime = S1TranslationRegime() == EL1;
            Enable = if IsEL1Regime then SCTLR_EL1.EnIA else SCTLR_EL2.EnIA;
            TrapEL2 = (EL2Enabled() && HCR_EL2.API == '0' &&
                      (HCR_EL2.TGE == '0' || HCR_EL2.E2H == '0'));
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL1
            Enable = SCTLR_EL1.EnIA;
            TrapEL2 = EL2Enabled() && HCR_EL2.API == '0';
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL2
            Enable = SCTLR_EL2.EnIA;
            TrapEL2 = FALSE;
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL3
            Enable = SCTLR_EL3.EnIA;
            TrapEL2 = FALSE;
            TrapEL3 = FALSE;

    if Enable == '0' then
        return x;
    elsif TrapEL3 && EL3SDDUndefPriority() then
        UNDEFINED;
    elsif TrapEL2 then
        TrapPACUse(EL2);
    elsif TrapEL3 then
        if EL3SDDUndef() then
            UNDEFINED;
        else
            TrapPACUse(EL3);
    else
        return Auth(x, y, APIAKey_EL1, FALSE, '0', is_combined);
```

Library pseudocode for aarch64/functions/pac/authib/AuthIB

```
// AuthIB()
// ========
// Returns a 64-bit value containing x, but replacing the pointer authentication code
// field bits with the extension of the address bits. The instruction checks a pointer
// authentication code in the pointer authentication code field bits of x, using the same
// algorithm and key as AddPACIB().

bits(64) AuthIB(bits(64) x, bits(64) y, boolean is_combined)
    boolean TrapEL2;
    boolean TrapEL3;
    bits(1)  Enable;
    bits(128) APIBKey_EL1;

    APIBKey_EL1 = APIBKeyHi_EL1<63:0> : APIBKeyLo_EL1<63:0>;
    case PSTATE.EL of
        when EL0
            boolean IsEL1Regime = S1TranslationRegime() == EL1;
            Enable = if IsEL1Regime then SCTLR_EL1.EnIB else SCTLR_EL2.EnIB;
            TrapEL2 = (EL2Enabled() && HCR_EL2.API == '0' &&
                       (HCR_EL2.TGE == '0' || HCR_EL2.E2H == '0'));
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL1
            Enable = SCTLR_EL1.EnIB;
            TrapEL2 = EL2Enabled() && HCR_EL2.API == '0';
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL2
            Enable = SCTLR_EL2.EnIB;
            TrapEL2 = FALSE;
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL3
            Enable = SCTLR_EL3.EnIB;
            TrapEL2 = FALSE;
            TrapEL3 = FALSE;

    if Enable == '0' then
        return x;
    elsif TrapEL3 && EL3SDDUndefPriority() then
        UNDEFINED;
    elsif TrapEL2 then
        TrapPACUse(EL2);
    elsif TrapEL3 then
        if EL3SDDUndef() then
            UNDEFINED;
        else
            TrapPACUse(EL3);
    else
        return Auth(x, y, APIBKey_EL1, FALSE, '1', is_combined);
```

Library pseudocode for aarch64/functions/pac/calcbottompacbit/AArch64.PACEffectiveTxSZ

```
// AArch64.PACEffectiveTxSZ()
// ==========================
// Compute the effective value for TxSZ used to determine the placement of the PAC field

bits(6) AArch64.PACEffectiveTxSZ(Regime regime, S1TTWParams walkparams)
    constant integer s1maxtxsz = AArch64.MaxTxSZ(walkparams.tgx);
    constant integer s1mintxsz = AArch64.S1MinTxSZ(regime, walkparams.d128,
                                               walkparams.ds, walkparams.tgx);

    if AArch64.S1TxSZFaults(regime, walkparams) then
        if ConstrainUnpredictable(Unpredictable_RESTnSZ) == Constraint_FORCE then
            if UInt(walkparams.txsz) < s1mintxsz then
                return s1mintxsz<5:0>;
            if UInt(walkparams.txsz) > s1maxtxsz then
                return s1maxtxsz<5:0>;
    elsif UInt(walkparams.txsz) < s1mintxsz then
        return s1mintxsz<5:0>;
    elsif UInt(walkparams.txsz) > s1maxtxsz then
        return s1maxtxsz<5:0>;

    return walkparams.txsz;
```

Library pseudocode for aarch64/functions/pac/calcbottompacbit/CalculateBottomPACBit

```
// CalculateBottomPACBit()
// =======================

integer CalculateBottomPACBit(bit top_bit)
    Regime regime;
    S1TTWParams walkparams;
    integer bottom_PAC_bit;

    regime = TranslationRegime(PSTATE.EL);
    ss = CurrentSecurityState();
    walkparams = AArch64.GetS1TTWParams(regime, ss, Replicate(top_bit, 64));
    bottom_PAC_bit = 64 - UInt(AArch64.PACEffectiveTxSZ(regime, walkparams));

    return bottom_PAC_bit;
```

Library pseudocode for aarch64/functions/pac/computepac/ComputePAC

```
// ComputePAC()
// ============

bits(64) ComputePAC(bits(64) data, bits(64) modifier, bits(64) key0, bits(64) key1,
                    boolean isgeneric)
    if UsePACIMP(isgeneric) then
        return ComputePACIMPDEF(data, modifier, key0, key1);
    if UsePACQARMA3(isgeneric) then
        boolean isqarma3 = TRUE;
        return ComputePACQARMA(data, modifier, key0, key1, isqarma3);
    if UsePACQARMA5(isgeneric) then
        boolean isqarma3 = FALSE;
        return ComputePACQARMA(data, modifier, key0, key1, isqarma3);
```

Library pseudocode for aarch64/functions/pac/computepac/ComputePACIMPDEF

```
// ComputePACIMPDEF()
// ==================
// Compute IMPLEMENTATION DEFINED cryptographic algorithm to be used for PAC calculation.

bits(64) ComputePACIMPDEF(bits(64) data, bits(64) modifier, bits(64) key0, bits(64) key1);
```

Library pseudocode for aarch64/functions/pac/computepac/ComputePACQARMA

```
// ComputePACQARMA()
// =================
// Compute QARMA3 or QARMA5 cryptographic algorithm for PAC calculation

bits(64) ComputePACQARMA(bits(64) data, bits(64) modifier, bits(64) key0,
                         bits(64) key1, boolean isqarma3)
    bits(64)  workingval;
    bits(64)  runningmod;
    bits(64)  roundkey;
    bits(64)  modk0;
    constant bits(64) Alpha = 0xC0AC29B7C97C50DD<63:0>;

    integer iterations;
    RC[0] = 0x0000000000000000<63:0>;
    RC[1] = 0x13198A2E03707344<63:0>;
    RC[2] = 0xA4093822299F31D0<63:0>;

    if isqarma3 then
        iterations = 2;
    else // QARMA5
        iterations = 4;
        RC[3] = 0x082EFA98EC4E6C89<63:0>;
        RC[4] = 0x452821E638D01377<63:0>;

    modk0 = key0<0>:key0<63:2>:(key0<63> EOR key0<1>);
    runningmod = modifier;
    workingval = data EOR key0;

    for i = 0 to iterations
        roundkey = key1 EOR runningmod;
        workingval  = workingval EOR roundkey;
        workingval = workingval EOR RC[i];
        if i > 0 then
            workingval = PACCellShuffle(workingval);
            workingval = PACMult(workingval);
        if isqarma3 then
            workingval = PACSub1(workingval);
        else
            workingval = PACSub(workingval);
        runningmod = TweakShuffle(runningmod<63:0>);
    roundkey = modk0 EOR runningmod;
    workingval = workingval EOR roundkey;
    workingval = PACCellShuffle(workingval);
    workingval = PACMult(workingval);
    if isqarma3 then
        workingval = PACSub1(workingval);
    else
        workingval = PACSub(workingval);
    workingval = PACCellShuffle(workingval);
    workingval = PACMult(workingval);
    workingval = key1 EOR workingval;
    workingval = PACCellInvShuffle(workingval);
    if isqarma3 then
        workingval = PACSub1(workingval);
    else
        workingval = PACInvSub(workingval);
    workingval = PACMult(workingval);
    workingval = PACCellInvShuffle(workingval);
    workingval = workingval EOR key0;
    workingval = workingval EOR runningmod;
    for i = 0 to iterations
        if isqarma3 then
            workingval = PACSub1(workingval);
        else
            workingval = PACInvSub(workingval);
        if i < iterations then
            workingval = PACMult(workingval);
            workingval = PACCellInvShuffle(workingval);
        runningmod = TweakInvShuffle(runningmod<63:0>);
        roundkey = key1 EOR runningmod;
```

```
        workingval = workingval EOR RC[iterations-i];
        workingval = workingval EOR roundkey;
        workingval = workingval EOR Alpha;
    workingval = workingval EOR modk0;

    return workingval;
```

Library pseudocode for aarch64/functions/pac/computepac/PACCellInvShuffle

```
// PACCellInvShuffle()
// ===================

bits(64) PACCellInvShuffle(bits(64) indata)
    bits(64) outdata;
    outdata<3:0> = indata<15:12>;
    outdata<7:4> = indata<27:24>;
    outdata<11:8> = indata<51:48>;
    outdata<15:12> = indata<39:36>;
    outdata<19:16> = indata<59:56>;
    outdata<23:20> = indata<47:44>;
    outdata<27:24> = indata<7:4>;
    outdata<31:28> = indata<19:16>;
    outdata<35:32> = indata<35:32>;
    outdata<39:36> = indata<55:52>;
    outdata<43:40> = indata<31:28>;
    outdata<47:44> = indata<11:8>;
    outdata<51:48> = indata<23:20>;
    outdata<55:52> = indata<3:0>;
    outdata<59:56> = indata<43:40>;
    outdata<63:60> = indata<63:60>;
    return outdata;
```

Library pseudocode for aarch64/functions/pac/computepac/PACCellShuffle

```
// PACCellShuffle()
// ================

bits(64) PACCellShuffle(bits(64) indata)
    bits(64) outdata;
    outdata<3:0> = indata<55:52>;
    outdata<7:4> = indata<27:24>;
    outdata<11:8> = indata<47:44>;
    outdata<15:12> = indata<3:0>;
    outdata<19:16> = indata<31:28>;
    outdata<23:20> = indata<51:48>;
    outdata<27:24> = indata<7:4>;
    outdata<31:28> = indata<43:40>;
    outdata<35:32> = indata<35:32>;
    outdata<39:36> = indata<15:12>;
    outdata<43:40> = indata<59:56>;
    outdata<47:44> = indata<23:20>;
    outdata<51:48> = indata<11:8>;
    outdata<55:52> = indata<39:36>;
    outdata<59:56> = indata<19:16>;
    outdata<63:60> = indata<63:60>;
    return outdata;
```

Library pseudocode for aarch64/functions/pac/computepac/PACInvSub

```
// PACInvSub()
// ===========

bits(64) PACInvSub(bits(64) Tinput)
    // This is a 4-bit substitution from the PRINCE-family cipher
    bits(64) Toutput;
    for i = 0 to 15
        case Tinput<4*i+3:4*i> of
            when '0000'  Toutput<4*i+3:4*i> = '0101';
            when '0001'  Toutput<4*i+3:4*i> = '1110';
            when '0010'  Toutput<4*i+3:4*i> = '1101';
            when '0011'  Toutput<4*i+3:4*i> = '1000';
            when '0100'  Toutput<4*i+3:4*i> = '1010';
            when '0101'  Toutput<4*i+3:4*i> = '1011';
            when '0110'  Toutput<4*i+3:4*i> = '0001';
            when '0111'  Toutput<4*i+3:4*i> = '1001';
            when '1000'  Toutput<4*i+3:4*i> = '0010';
            when '1001'  Toutput<4*i+3:4*i> = '0110';
            when '1010'  Toutput<4*i+3:4*i> = '1111';
            when '1011'  Toutput<4*i+3:4*i> = '0000';
            when '1100'  Toutput<4*i+3:4*i> = '0100';
            when '1101'  Toutput<4*i+3:4*i> = '1100';
            when '1110'  Toutput<4*i+3:4*i> = '0111';
            when '1111'  Toutput<4*i+3:4*i> = '0011';
    return Toutput;
```

Library pseudocode for aarch64/functions/pac/computepac/PACMult

```
// PACMult()
// =========

bits(64) PACMult(bits(64) Sinput)
    bits(4)  t0;
    bits(4)  t1;
    bits(4)  t2;
    bits(4)  t3;
    bits(64) Soutput;

    for i = 0 to 3
        t0<3:0> = RotCell(Sinput<4*(i+8)+3:4*(i+8)>, 1) EOR RotCell(Sinput<4*(i+4)+3:4*(i+4)>, 2);
        t0<3:0> = t0<3:0> EOR RotCell(Sinput<4*(i)+3:4*(i)>, 1);
        t1<3:0> = RotCell(Sinput<4*(i+12)+3:4*(i+12)>, 1) EOR RotCell(Sinput<4*(i+4)+3:4*(i+4)>, 1);
        t1<3:0> = t1<3:0> EOR RotCell(Sinput<4*(i)+3:4*(i)>, 2);
        t2<3:0> = RotCell(Sinput<4*(i+12)+3:4*(i+12)>, 2) EOR RotCell(Sinput<4*(i+8)+3:4*(i+8)>, 1);
        t2<3:0> = t2<3:0> EOR RotCell(Sinput<4*(i)+3:4*(i)>, 1);
        t3<3:0> = RotCell(Sinput<4*(i+12)+3:4*(i+12)>, 1) EOR RotCell(Sinput<4*(i+8)+3:4*(i+8)>, 2);
        t3<3:0> = t3<3:0> EOR RotCell(Sinput<4*(i+4)+3:4*(i+4)>, 1);
        Soutput<4*i+3:4*i> = t3<3:0>;
        Soutput<4*(i+4)+3:4*(i+4)> = t2<3:0>;
        Soutput<4*(i+8)+3:4*(i+8)> = t1<3:0>;
        Soutput<4*(i+12)+3:4*(i+12)> = t0<3:0>;
    return Soutput;
```

Library pseudocode for aarch64/functions/pac/computepac/PACSub

```
// PACSub()
// ========

bits(64) PACSub(bits(64) Tinput)
    // This is a 4-bit substitution from the PRINCE-family cipher
    bits(64) Toutput;
    for i = 0 to 15
        case Tinput<4*i+3:4*i> of
            when '0000'  Toutput<4*i+3:4*i> = '1011';
            when '0001'  Toutput<4*i+3:4*i> = '0110';
            when '0010'  Toutput<4*i+3:4*i> = '1000';
            when '0011'  Toutput<4*i+3:4*i> = '1111';
            when '0100'  Toutput<4*i+3:4*i> = '1100';
            when '0101'  Toutput<4*i+3:4*i> = '0000';
            when '0110'  Toutput<4*i+3:4*i> = '1001';
            when '0111'  Toutput<4*i+3:4*i> = '1110';
            when '1000'  Toutput<4*i+3:4*i> = '0011';
            when '1001'  Toutput<4*i+3:4*i> = '0111';
            when '1010'  Toutput<4*i+3:4*i> = '0100';
            when '1011'  Toutput<4*i+3:4*i> = '0101';
            when '1100'  Toutput<4*i+3:4*i> = '1101';
            when '1101'  Toutput<4*i+3:4*i> = '0010';
            when '1110'  Toutput<4*i+3:4*i> = '0001';
            when '1111'  Toutput<4*i+3:4*i> = '1010';
    return Toutput;
```

Library pseudocode for aarch64/functions/pac/computepac/PacSub1

```
// PacSub1()
// =========

bits(64) PACSub1(bits(64) Tinput)
    // This is a 4-bit substitution from Qarma sigma1
    bits(64) Toutput;
    for i = 0 to 15
        case Tinput<4*i+3:4*i> of
            when '0000' Toutput<4*i+3:4*i> = '1010';
            when '0001' Toutput<4*i+3:4*i> = '1101';
            when '0010' Toutput<4*i+3:4*i> = '1110';
            when '0011' Toutput<4*i+3:4*i> = '0110';
            when '0100' Toutput<4*i+3:4*i> = '1111';
            when '0101' Toutput<4*i+3:4*i> = '0111';
            when '0110' Toutput<4*i+3:4*i> = '0011';
            when '0111' Toutput<4*i+3:4*i> = '0101';
            when '1000' Toutput<4*i+3:4*i> = '1001';
            when '1001' Toutput<4*i+3:4*i> = '1000';
            when '1010' Toutput<4*i+3:4*i> = '0000';
            when '1011' Toutput<4*i+3:4*i> = '1100';
            when '1100' Toutput<4*i+3:4*i> = '1011';
            when '1101' Toutput<4*i+3:4*i> = '0001';
            when '1110' Toutput<4*i+3:4*i> = '0010';
            when '1111' Toutput<4*i+3:4*i> = '0100';
    return Toutput;
```

Library pseudocode for aarch64/functions/pac/computepac/RC

```
// RC[]
// ====

array bits(64) RC[0..4];
```

Library pseudocode for aarch64/functions/pac/computepac/RotCell

```
// RotCell()
// =========

bits(4) RotCell(bits(4) incell, integer amount)
    bits(8) tmp;
    bits(4) outcell;

    // assert amount>3 || amount<1;
    tmp<7:0> = incell<3:0>:incell<3:0>;
    outcell = tmp<7-amount:4-amount>;
    return outcell;
```

Library pseudocode for aarch64/functions/pac/computepac/TweakCellInvRot

```
// TweakCellInvRot()
// =================

bits(4) TweakCellInvRot(bits(4) incell)
    bits(4) outcell;
    outcell<3> = incell<2>;
    outcell<2> = incell<1>;
    outcell<1> = incell<0>;
    outcell<0> = incell<0> EOR incell<3>;
    return outcell;
```

Library pseudocode for aarch64/functions/pac/computepac/TweakCellRot

```
// TweakCellRot()
// ==============

bits(4) TweakCellRot(bits(4) incell)
    bits(4) outcell;
    outcell<3> = incell<0> EOR incell<1>;
    outcell<2> = incell<3>;
    outcell<1> = incell<2>;
    outcell<0> = incell<1>;
    return outcell;
```

Library pseudocode for aarch64/functions/pac/computepac/TweakInvShuffle

```
// TweakInvShuffle()
// =================

bits(64) TweakInvShuffle(bits(64) indata)
    bits(64) outdata;
    outdata<3:0> = TweakCellInvRot(indata<51:48>);
    outdata<7:4> = indata<55:52>;
    outdata<11:8> = indata<23:20>;
    outdata<15:12> = indata<27:24>;
    outdata<19:16> = indata<3:0>;
    outdata<23:20> = indata<7:4>;
    outdata<27:24> = TweakCellInvRot(indata<11:8>);
    outdata<31:28> = indata<15:12>;
    outdata<35:32> = TweakCellInvRot(indata<31:28>);
    outdata<39:36> = TweakCellInvRot(indata<63:60>);
    outdata<43:40> = TweakCellInvRot(indata<59:56>);
    outdata<47:44> = TweakCellInvRot(indata<19:16>);
    outdata<51:48> = indata<35:32>;
    outdata<55:52> = indata<39:36>;
    outdata<59:56> = indata<43:40>;
    outdata<63:60> = TweakCellInvRot(indata<47:44>);
    return outdata;
```

Library pseudocode for aarch64/functions/pac/computepac/TweakShuffle

```
// TweakShuffle()
// ==============

bits(64) TweakShuffle(bits(64) indata)
    bits(64) outdata;
    outdata<3:0> = indata<19:16>;
    outdata<7:4> = indata<23:20>;
    outdata<11:8> = TweakCellRot(indata<27:24>);
    outdata<15:12> = indata<31:28>;
    outdata<19:16> = TweakCellRot(indata<47:44>);
    outdata<23:20> = indata<11:8>;
    outdata<27:24> = indata<15:12>;
    outdata<31:28> = TweakCellRot(indata<35:32>);
    outdata<35:32> = indata<51:48>;
    outdata<39:36> = indata<55:52>;
    outdata<43:40> = indata<59:56>;
    outdata<47:44> = TweakCellRot(indata<63:60>);
    outdata<51:48> = TweakCellRot(indata<3:0>);
    outdata<55:52> = indata<7:4>;
    outdata<59:56> = TweakCellRot(indata<43:40>);
    outdata<63:60> = TweakCellRot(indata<39:36>);
    return outdata;
```

Library pseudocode for aarch64/functions/pac/computepac/UsePACIMP

```
// UsePACIMP()
// ===========
// Checks whether IMPLEMENTATION DEFINED cryptographic algorithm to be used for PAC
// calculation.

boolean UsePACIMP(boolean isgeneric)
    return if isgeneric then HavePACIMPGeneric() else HavePACIMPAuth();
```

Library pseudocode for aarch64/functions/pac/computepac/UsePACQARMA3

```
// UsePACQARMA3()
// ==============
// Checks whether QARMA3 cryptographic algorithm to be used for PAC calculation.

boolean UsePACQARMA3(boolean isgeneric)
    return if isgeneric then HavePACQARMA3Generic() else HavePACQARMA3Auth();
```

Library pseudocode for aarch64/functions/pac/computepac/UsePACQARMA5

```
// UsePACQARMA5()
// ==============
// Checks whether QARMA5 cryptographic algorithm to be used for PAC calculation.

boolean UsePACQARMA5(boolean isgeneric)
    return if isgeneric then HavePACQARMA5Generic() else HavePACQARMA5Auth();
```

Library pseudocode for aarch64/functions/pac/pac/ConstPACField

```
// ConstPACField()
// ===============
// Returns TRUE if bit<55> can be used to determine the size of the PAC field, FALSE otherwise.

boolean ConstPACField()
    return IsFeatureImplemented(FEAT_CONSTPACFIELD);
```

Library pseudocode for aarch64/functions/pac/pac/HaveEnhancedPAC

```
// HaveEnhancedPAC()
// =================
// Returns TRUE if support for EnhancedPAC is implemented, FALSE otherwise.

boolean HaveEnhancedPAC()
    return IsFeatureImplemented(FEAT_EPAC);
```

Library pseudocode for aarch64/functions/pac/pac/HaveEnhancedPAC2

```
// HaveEnhancedPAC2()
// ==================
// Returns TRUE if support for EnhancedPAC2 is implemented, FALSE otherwise.

boolean HaveEnhancedPAC2()
    return IsFeatureImplemented(FEAT_PAuth2);
```

Library pseudocode for aarch64/functions/pac/pac/HaveFPAC

```
// HaveFPAC()
// ==========
// Returns TRUE if support for FPAC is implemented, FALSE otherwise.

boolean HaveFPAC()
    return IsFeatureImplemented(FEAT_FPAC);
```

Library pseudocode for aarch64/functions/pac/pac/HaveFPACCombined

```
// HaveFPACCombined()
// ==================
// Returns TRUE if support for FPACCombined is implemented, FALSE otherwise.

boolean HaveFPACCombined()
    return IsFeatureImplemented(FEAT_FPACCOMBINE);
```

Library pseudocode for aarch64/functions/pac/pac/HavePACExt

```
// HavePACExt()
// ============
// Returns TRUE if support for the PAC extension is implemented, FALSE otherwise.

boolean HavePACExt()
    return IsFeatureImplemented(FEAT_PAuth);
```

Library pseudocode for aarch64/functions/pac/pac/HavePACIMPAuth

```
// HavePACIMPAuth()
// ================
// Returns TRUE if support for PAC IMP Auth is implemented, FALSE otherwise.

boolean HavePACIMPAuth()
    return IsFeatureImplemented(FEAT_PACIMP);
```

Library pseudocode for aarch64/functions/pac/pac/HavePACIMPGeneric

```
// HavePACIMPGeneric()
// ===================
// Returns TRUE if support for PAC IMP Generic is implemented, FALSE otherwise.

boolean HavePACIMPGeneric()
    return IsFeatureImplemented(FEAT_PACIMP);
```

Library pseudocode for aarch64/functions/pac/pac/HavePACQARMA3Auth

```
// HavePACQARMA3Auth()
// ===================
// Returns TRUE if support for PAC QARMA3 Auth is implemented, FALSE otherwise.

boolean HavePACQARMA3Auth()
    return IsFeatureImplemented(FEAT_PACQARMA3);
```

Library pseudocode for aarch64/functions/pac/pac/HavePACQARMA3Generic

```
// HavePACQARMA3Generic()
// ======================
// Returns TRUE if support for PAC QARMA3 Generic is implemented, FALSE otherwise.

boolean HavePACQARMA3Generic()
    return IsFeatureImplemented(FEAT_PACQARMA3);
```

Library pseudocode for aarch64/functions/pac/pac/HavePACQARMA5Auth

```
// HavePACQARMA5Auth()
// ===================
// Returns TRUE if support for PAC QARMA5 Auth is implemented, FALSE otherwise.

boolean HavePACQARMA5Auth()
    return IsFeatureImplemented(FEAT_PACQARMA5);
```

Library pseudocode for aarch64/functions/pac/pac/HavePACQARMA5Generic

```
// HavePACQARMA5Generic()
// ======================
// Returns TRUE if support for PAC QARMA5 Generic is implemented, FALSE otherwise.

boolean HavePACQARMA5Generic()
    return IsFeatureImplemented(FEAT_PACQARMA5);
```

Library pseudocode for aarch64/functions/pac/pac/PtrHasUpperAndLowerAddRanges

```
// PtrHasUpperAndLowerAddRanges()
// ==============================
// Returns TRUE if the pointer has upper and lower address ranges, FALSE otherwise.

boolean PtrHasUpperAndLowerAddRanges()
    regime = TranslationRegime(PSTATE.EL);
    return HasUnprivileged(regime);
```

Library pseudocode for aarch64/functions/pac/strip/Strip

```
// Strip()
// =======
// Strip() returns a 64-bit value containing A, but replacing the pointer authentication
// code field bits with the extension of the address bits. This can apply to either
// instructions or data, where, as the use of tagged pointers is distinct, it might be
// handled differently.

bits(64) Strip(bits(64) A, boolean data)
    bits(64) original_ptr;
    bits(64) extfield;
    boolean tbi = EffectiveTBI(A, !data, PSTATE.EL) == '1';
    boolean mtx = EffectiveMTX(A, !data, PSTATE.EL) == '1';
    integer bottom_PAC_bit = CalculateBottomPACBit(A<55>);
    extfield = Replicate(A<55>, 64);

    // If the VA is 56 or 55 bits and Top Byte is Ignored,
    // there are no unused bits left for the PAC
    if tbi && bottom_PAC_bit >= 55 then
        return A;

    if tbi then
        original_ptr = (A<63:56> :
                        extfield<(56-bottom_PAC_bit)-1:0> : A<bottom_PAC_bit-1:0>);
    elsif mtx then
        original_ptr = (extfield<63:60> : A<59:56> :
                        extfield<(56-bottom_PAC_bit)-1:0> : A<bottom_PAC_bit-1:0>);
    else
        original_ptr =  extfield<(64-bottom_PAC_bit)-1:0> : A<bottom_PAC_bit-1:0>;

    return original_ptr;
```

Library pseudocode for aarch64/functions/pac/trappacuse/TrapPACUse

```
// TrapPACUse()
// ============
// Used for the trapping of the pointer authentication functions by higher exception
// levels.

TrapPACUse(bits(2) target_el)
    assert HaveEL(target_el) && target_el !=  EL0 && UInt(target_el) >= UInt(PSTATE.EL);

    bits(64) preferred_exception_return = ThisInstrAddr(64);
    ExceptionRecord except;
    vect_offset = 0;
    except = ExceptionSyndrome(Exception_PACTrap);
    AArch64.TakeException(target_el, except, preferred_exception_return, vect_offset);
```

Library pseudocode for aarch64/functions/predictionrestrict/AArch64.RestrictPrediction

```
// AArch64.RestrictPrediction()
// ============================
// Clear all predictions in the context.

AArch64.RestrictPrediction(bits(64) val, RestrictType restriction)

    ExecutionCntxt c;
    target_el    = val<25:24>;

    // If the target EL is not implemented or the instruction is executed at an
    // EL lower than the specified level, the instruction is treated as a NOP.
    if !HaveEL(target_el) || UInt(target_el) > UInt(PSTATE.EL) then EndOfInstruction();

    bit ns  = val<26>;
    bit nse = val<27>;
    ss = TargetSecurityState(ns, nse);

    // If the combination of Security state and Exception level is not implemented,
    // the instruction is treated as a NOP.
    if ss == SS_Root && target_el != EL3 then EndOfInstruction();
    if !HaveRME() && target_el == EL3 && ss != SS_Secure then EndOfInstruction();

    c.security  = ss;
    c.target_el = target_el;

    if EL2Enabled() then
        if (PSTATE.EL == EL0 && !IsInHost()) || PSTATE.EL == EL1 then
            c.is_vmid_valid = TRUE;
            c.all_vmid      = FALSE;
            c.vmid          = VMID[];

        elsif (target_el == EL0 && !ELIsInHost(target_el)) || target_el == EL1 then
            c.is_vmid_valid = TRUE;
            c.all_vmid      = val<48> == '1';
            c.vmid          = val<47:32>;       // Only valid if  val<48> == '0';

        else
            c.is_vmid_valid = FALSE;
    else
        c.is_vmid_valid = FALSE;

    if PSTATE.EL == EL0 then
        c.is_asid_valid = TRUE;
        c.all_asid      = FALSE;
        c.asid          = ASID[];

    elsif target_el == EL0 then
        c.is_asid_valid = TRUE;
        c.all_asid      = val<16> == '1';
        c.asid          = val<15:0>;            // Only valid if  val<16> == '0';

    else
        c.is_asid_valid = FALSE;

    c.restriction = restriction;
    RESTRICT_PREDICTIONS(c);
```

Library pseudocode for aarch64/functions/prefetch/Prefetch

```
// Prefetch()
// ==========

// Decode and execute the prefetch hint on ADDRESS specified by PRFOP

Prefetch(bits(64) address, bits(5) prfop)
    PrefetchHint hint;
    integer target;
    boolean stream;

    case prfop<4:3> of
        when '00' hint = Prefetch_READ;         // PLD: prefetch for load
        when '01' hint = Prefetch_EXEC;         // PLI: preload instructions
        when '10' hint = Prefetch_WRITE;        // PST: prepare for store
        when '11' return;                       // unallocated hint
    target = UInt(prfop<2:1>);                  // target cache level
    stream = (prfop<0> != '0');                 // streaming (non-temporal)
    Hint_Prefetch(address, hint, target, stream);
    return;
```

Library pseudocode for aarch64/functions/pstatefield/PSTATEField

```
// PSTATEField
// ===========
// MSR (immediate) instruction destinations.

enumeration PSTATEField {PSTATEField_DAIFSet, PSTATEField_DAIFClr,
                         PSTATEField_PAN, // Armv8.1
                         PSTATEField_UAO, // Armv8.2
                         PSTATEField_DIT, // Armv8.4
                         PSTATEField_SSBS,
                         PSTATEField_TCO, // Armv8.5
                         PSTATEField_SVCRSM,
                         PSTATEField_SVCRZA,
                         PSTATEField_SVCRSMZA,
                         PSTATEField_ALLINT,
                         PSTATEField_PM,
                         PSTATEField_SP
                         };
```

Library pseudocode for aarch64/functions/ras/AArch64.ESBOperation

```
// AArch64.ESBOperation()
// ======================
// Perform the AArch64 ESB operation, either for ESB executed in AArch64 state, or for
// ESB in AArch32 state when SError interrupts are routed to an Exception level using
// AArch64

AArch64.ESBOperation()
    bits(2) target_el;
    boolean masked;

    (masked, target_el) = AArch64.PhysicalSErrorTarget();

    intdis = Halted() || ExternalDebugInterruptsDisabled(target_el);
    masked = masked || intdis;

    // Check for a masked Physical SError pending that can be synchronized
    // by an Error synchronization event.
    if masked && IsSynchronizablePhysicalSErrorPending() then
        // This function might be called for an interworking case, and INTdis is masking
        // the SError interrupt.
        if ELUsingAArch32(S1TranslationRegime()) then
            bits(32) syndrome = Zeros(32);
            syndrome<31> = '1'; // A
            syndrome<15:0> = AArch32.PhysicalSErrorSyndrome();
            DISR = syndrome;
        else
            implicit_esb = FALSE;
            bits(64) syndrome = Zeros(64);
            syndrome<31> = '1'; // A
            syndrome<24:0> = AArch64.PhysicalSErrorSyndrome(implicit_esb);
            DISR_EL1 = syndrome;
        ClearPendingPhysicalSError();               // Set ISR_EL1.A to 0

    return;
```

Library pseudocode for aarch64/functions/ras/AArch64.EncodeAsyncErrorSyndrome

```
// AArch64.EncodeAsyncErrorSyndrome()
// ==================================
// Return the encoding for corresponding ErrorState.

bits(3) AArch64.EncodeAsyncErrorSyndrome(ErrorState errorstate)
    case errorstate of
        when ErrorState_UC  return '000';
        when ErrorState_UEU return '001';
        when ErrorState_UEO return '010';
        when ErrorState_UER return '011';
        when ErrorState_CE  return '110';
        otherwise Unreachable();
```

Library pseudocode for aarch64/functions/ras/AArch64.EncodeSyncErrorSyndrome

```
// AArch64.EncodeSyncErrorSyndrome()
// =================================
// Return the encoding for corresponding ErrorState.

bits(2) AArch64.EncodeSyncErrorSyndrome(ErrorState errorstate)
    case errorstate of
        when ErrorState_UC  return '10';
        when ErrorState_UEO return '11';
        when ErrorState_UER return '00';
        otherwise Unreachable();
```

Library pseudocode for aarch64/functions/ras/AArch64.PhysicalSErrorSyndrome

```
// AArch64.PhysicalSErrorSyndrome()
// ================================
// Generate SError syndrome.

bits(25) AArch64.PhysicalSErrorSyndrome(boolean implicit_esb)
    bits(25) syndrome = Zeros(25);
    FaultRecord fault = GetPendingPhysicalSError();
    ErrorState errorstate = AArch64.PEErrorState(fault);
    if errorstate == ErrorState_Uncategorized then
        syndrome = Zeros(25);
    elsif errorstate == ErrorState_IMPDEF then
        syndrome<24> = '1';                                          // IDS
        syndrome<23:0> = bits(24) IMPLEMENTATION_DEFINED "IMPDEF ErrorState";
    else
        syndrome<24> = '0';                                          // IDS
        syndrome<13> = (if implicit_esb then '1' else '0');          // IESB
        syndrome<12:10> = AArch64.EncodeAsyncErrorSyndrome(errorstate); // AET
        syndrome<9> = fault.extflag;                                 // EA
        syndrome<5:0> = '010001';                                    // DFSC
    return syndrome;
```

Library pseudocode for aarch64/functions/ras/AArch64.vESBOperation

```
// AArch64.vESBOperation()
// =======================
// Perform the AArch64 ESB operation for virtual SError interrupts, either for ESB
// executed in AArch64 state, or for ESB in AArch32 state with EL2 using AArch64 state

AArch64.vESBOperation()
    assert PSTATE.EL IN {EL0, EL1} && EL2Enabled();

    // If physical SError interrupts are routed to EL2, and TGE is not set, then a virtual
    // SError interrupt might be pending
    vSEI_enabled = HCR_EL2.TGE == '0' && HCR_EL2.AMO == '1';
    vSEI_pending = vSEI_enabled && HCR_EL2.VSE == '1';
    vintdis      = Halted() || ExternalDebugInterruptsDisabled(EL1);
    vmasked      = vintdis || PSTATE.A == '1';

    // Check for a masked virtual SError pending
    if vSEI_pending && vmasked then
        // This function might be called for the interworking case, and INTdis is masking
        // the virtual SError interrupt.
        if ELUsingAArch32(EL1) then
            bits(32) target = Zeros(32);
            target<31> = '1';              // A
            target<15:14> = VDFSR<15:14>;  // AET
            target<12>    = VDFSR<12>;     // ExT
            target<9>     = TTBCR.EAE;     // LPAE
            if TTBCR.EAE == '1' then       // Long-descriptor format
                target<5:0>    = '010001'; // STATUS
            else                           // Short-descriptor format
                target<10,3:0> = '10110';  // FS
            VDISR = target;
        else
            bits(64) target = Zeros(64);
            target<31> = '1';              // A
            target<24:0> = VSESR_EL2<24:0>;
            VDISR_EL2 = target;
        HCR_EL2.VSE = '0';                       // Clear pending virtual SError

    return;
```

Library pseudocode for aarch64/functions/ras/FirstRecordOfNode

```
// FirstRecordOfNode()
// ===================
// Return the first record in the node that contains the record n.

integer FirstRecordOfNode(integer n)
    for q = n downto 0
        if IsFirstRecordOfNode(q) then return q;
    Unreachable();
```

Library pseudocode for aarch64/functions/ras/IsCommonFaultInjectionImplemented

```
// IsCommonFaultInjectionImplemented()
// ===================================
// Check if the Common Fault Injection Model Extension is implemented by the node that owns this
// error record.

boolean IsCommonFaultInjectionImplemented(integer n);
```

Library pseudocode for aarch64/functions/ras/IsCountableErrorsRecorded

```
// IsCountableErrorsRecorded()
// ===========================
// Check whether Error record n records countable errors.

boolean IsCountableErrorsRecorded(integer n);
```

Library pseudocode for aarch64/functions/ras/IsErrorAddressIncluded

```
// IsErrorAddressIncluded()
// ========================
// Check whether Error record n includes an address associated with an error.

boolean IsErrorAddressIncluded(integer n);
```

Library pseudocode for aarch64/functions/ras/IsErrorRecordImplemented

```
// IsErrorRecordImplemented()
// ==========================
// Is the error record n implemented

boolean IsErrorRecordImplemented(integer n);
```

Library pseudocode for aarch64/functions/ras/IsFirstRecordOfNode

```
// IsFirstRecordOfNode()
// =====================
// Check if the record q is the first error record in its node.

boolean IsFirstRecordOfNode(integer q);
```

Library pseudocode for aarch64/functions/ras/IsSPMUCounterImplemented

```
// IsSPMUCounterImplemented()
// ==========================
// Does the System PMU s implement the counter n.

boolean IsSPMUCounterImplemented(integer s, integer n);
```

Library pseudocode for aarch64/functions/rcw/ProtectionEnabled

```
// ProtectionEnabled()
// ===================
// Returns TRUE if the ProtectedBit is
// enabled in the current Exception level.

boolean ProtectionEnabled(bits(2) el)
    assert HaveEL(el);
    regime = S1TranslationRegime(el);
    assert(!ELUsingAArch32(regime));
    if (!IsD128Enabled(el)) then
        case regime of
            when EL1
                return IsTCR2EL1Enabled() && TCR2_EL1.PnCH == '1';
            when EL2
                return IsTCR2EL2Enabled() && TCR2_EL2.PnCH == '1';
            when EL3
                return TCR_EL3.PnCH == '1';
    else
        return TRUE;
    return FALSE;
```

Library pseudocode for aarch64/functions/rcw/RCW128_PROTECTED_BIT

```
constant integer RCW128_PROTECTED_BIT = 114;
```

Library pseudocode for aarch64/functions/rcw/RCW64_PROTECTED_BIT

```
constant integer RCW64_PROTECTED_BIT = 52;
```

Library pseudocode for aarch64/functions/rcw/RCWCheck

```
// RCWCheck()
// ==========
// Returns nzcv based on : if the new value for RCW/RCWS instructions satisfy RCW and/or RCWS checks
// Z is set to 1 if RCW checks fail
// C is set to 0 if RCWS checks fail

bits(4) RCWCheck(bits(N) old, bits(N) new, boolean soft)
    assert N IN {64,128};
    integer protectedbit = if N == 128 then RCW128_PROTECTED_BIT else RCW64_PROTECTED_BIT;
    boolean rcw_fail = FALSE;
    boolean rcws_fail = FALSE;
    boolean rcw_state_fail = FALSE;
    boolean rcws_state_fail = FALSE;
    boolean rcw_mask_fail = FALSE;
    boolean rcws_mask_fail = FALSE;

    //Effective RCWMask calculation
    bits(N) rcwmask = RCWMASK_EL1<N-1:0>;
    if N == 64 then
        rcwmask<49:18> = Replicate(rcwmask<17>,32);
        rcwmask<0> = '0';
    else
        rcwmask<55:17> = Replicate(rcwmask<16>,39);
        rcwmask<126:125,120:119,107:101,90:56,1:0> = Zeros(48);

    //Effective RCWSMask calculation
    bits(N) rcwsoftmask = RCWSMASK_EL1<N-1:0>;
    if N == 64 then
        rcwsoftmask<49:18> = Replicate(rcwsoftmask<17>,32);
        rcwsoftmask<0> = '0';
        if(ProtectionEnabled(PSTATE.EL)) then
            rcwsoftmask<52> = '0';
    else
        rcwsoftmask<55:17> = Replicate(rcwsoftmask<16>,39);
        rcwsoftmask<126:125,120:119,107:101,90:56,1:0> = Zeros(48);
        rcwsoftmask<114> = '0';

    //RCW Checks
    //State Check
    if (ProtectionEnabled(PSTATE.EL)) then
        if old<protectedbit> == '1' then
            rcw_state_fail = new<protectedbit,0> != old<protectedbit,0>;
        elsif old<protectedbit> == '0' then
            rcw_state_fail = new<protectedbit> != old<protectedbit>;

    //Mask Check
    if (ProtectionEnabled(PSTATE.EL)) then
        if old<protectedbit,0> == '11' then
            rcw_mask_fail = !IsZero((new EOR old) AND NOT(rcwmask));

    //RCWS Checks
    if soft then
        //State Check
        if old<0> == '1' then
            rcws_state_fail = new<0> != old<0>;
        elsif (!ProtectionEnabled(PSTATE.EL) ||
              (ProtectionEnabled(PSTATE.EL) && old<protectedbit> == '0')) then
            rcws_state_fail = new<0> != old<0> ;
        //Mask Check
        if old<0> == '1' then
            rcws_mask_fail = !IsZero((new EOR old) AND NOT(rcwsoftmask));

    rcw_fail = rcw_state_fail  || rcw_mask_fail ;
    rcws_fail = rcws_state_fail || rcws_mask_fail;

    bit n = '0';
    bit z = if rcw_fail then '1' else '0';
    bit c = if rcws_fail then '0' else '1';
    bit v = '0';
    return <n, z, c, v>;
```

Library pseudocode for aarch64/functions/reduceop/Reduce

```
// Reduce()
// ========

bits(esize) Reduce(ReduceOp op, bits(N) input, integer esize)
    boolean altfp = HaveAltFP() && !UsingAArch32() && FPCR.AH == '1';
    return Reduce(op, input, esize, altfp);

// Reduce()
// ========
// Perform the operation 'op' on pairs of elements from the input vector,
// reducing the vector to a scalar result. The 'altfp' argument controls
// alternative floating-point behavior.

bits(esize) Reduce(ReduceOp op, bits(N) input, integer esize, boolean altfp)
    assert esize IN {8,16,32,64};
    integer half;
    bits(esize) hi;
    bits(esize) lo;
    bits(esize) result;

    if N == esize then
        return input<esize-1:0>;

    half = N DIV 2;
    hi = Reduce(op, input<N-1:half>, esize, altfp);
    lo = Reduce(op, input<half-1:0>, esize, altfp);

    case op of
        when ReduceOp_FMINNUM
            result = FPMinNum(lo, hi, FPCR[]);
        when ReduceOp_FMAXNUM
            result = FPMaxNum(lo, hi, FPCR[]);
        when ReduceOp_FMIN
            result = FPMin(lo, hi, FPCR[], altfp);
        when ReduceOp_FMAX
            result = FPMax(lo, hi, FPCR[], altfp);
        when ReduceOp_FADD
            result = FPAdd(lo, hi, FPCR[]);
        when ReduceOp_ADD
            result = lo + hi;

    return result;
```

Library pseudocode for aarch64/functions/reduceop/ReduceOp

```
// ReduceOp
// ========
// Vector reduce instruction types.

enumeration ReduceOp {ReduceOp_FMINNUM, ReduceOp_FMAXNUM,
                      ReduceOp_FMIN, ReduceOp_FMAX,
                      ReduceOp_FADD, ReduceOp_ADD};
```

Library pseudocode for aarch64/functions/registers/AArch64.MaybeZeroRegisterUppers

```
// AArch64.MaybeZeroRegisterUppers()
// =================================
// On taking an exception to  AArch64 from AArch32, it is CONSTRAINED UNPREDICTABLE whether the top
// 32 bits of registers visible at any lower Exception level using AArch32 are set to zero.

AArch64.MaybeZeroRegisterUppers()
    assert UsingAArch32();         // Always called from AArch32 state before entering AArch64 state

    integer first;
    integer last;
    boolean include_R15;
    if PSTATE.EL == EL0 && !ELUsingAArch32(EL1) then
        first = 0;  last = 14;  include_R15 = FALSE;
    elsif PSTATE.EL IN {EL0, EL1} && EL2Enabled() && !ELUsingAArch32(EL2) then
        first = 0;  last = 30;  include_R15 = FALSE;
    else
        first = 0;  last = 30;  include_R15 = TRUE;

    for n = first to last
        if (n != 15 || include_R15) && ConstrainUnpredictableBool(Unpredictable_ZEROUPPER) then
            _R[n]<63:32> = Zeros(32);

    return;
```

Library pseudocode for aarch64/functions/registers/AArch64.ResetGeneralRegisters

```
// AArch64.ResetGeneralRegisters()
// ===============================

AArch64.ResetGeneralRegisters()

    for i = 0 to 30
        X[i, 64] = bits(64) UNKNOWN;

    return;
```

Library pseudocode for aarch64/functions/registers/AArch64.ResetSIMDFPRegisters

```
// AArch64.ResetSIMDFPRegisters()
// ==============================

AArch64.ResetSIMDFPRegisters()

    for i = 0 to 31
        V[i, 128] = bits(128) UNKNOWN;

    return;
```

Library pseudocode for aarch64/functions/registers/AArch64.ResetSpecialRegisters

```
// AArch64.ResetSpecialRegisters()
// ===============================

AArch64.ResetSpecialRegisters()

    // AArch64 special registers
    SP_EL0 = bits(64) UNKNOWN;
    SP_EL1 = bits(64) UNKNOWN;
    SPSR_EL1 = bits(64) UNKNOWN;
    ELR_EL1  = bits(64) UNKNOWN;
    if HaveEL(EL2) then
        SP_EL2 = bits(64) UNKNOWN;
        SPSR_EL2 = bits(64) UNKNOWN;
        ELR_EL2  = bits(64) UNKNOWN;
    if HaveEL(EL3) then
        SP_EL3 = bits(64) UNKNOWN;
        SPSR_EL3 = bits(64) UNKNOWN;
        ELR_EL3  = bits(64) UNKNOWN;

    // AArch32 special registers that are not architecturally mapped to AArch64 registers
    if HaveAArch32EL(EL1) then
        SPSR_fiq<31:0> = bits(32) UNKNOWN;
        SPSR_irq<31:0> = bits(32) UNKNOWN;
        SPSR_abt<31:0> = bits(32) UNKNOWN;
        SPSR_und<31:0> = bits(32) UNKNOWN;

    // External debug special registers
    DLR_EL0 = bits(64) UNKNOWN;
    DSPSR_EL0 = bits(64) UNKNOWN;

    return;
```

Library pseudocode for aarch64/functions/registers/AArch64.ResetSystemRegisters

```
// AArch64.ResetSystemRegisters()
// ==============================

AArch64.ResetSystemRegisters(boolean cold_reset);
```

Library pseudocode for aarch64/functions/registers/PC

```
// PC - non-assignment form
// ========================
// Read program counter.

bits(64) PC[]
    return _PC;
```

Library pseudocode for aarch64/functions/registers/SP

```
// SP[] - assignment form
// ======================
// Write to stack pointer from a 64-bit value.

SP[] = bits(64) value
    if PSTATE.SP == '0' then
        SP_EL0 = value;
    else
        case PSTATE.EL of
            when EL0  SP_EL0 = value;
            when EL1  SP_EL1 = value;
            when EL2  SP_EL2 = value;
            when EL3  SP_EL3 = value;
    return;

// SP[] - non-assignment form
// ==========================
// Read stack pointer with slice of 64 bits.

bits(64) SP[]
    if PSTATE.SP == '0' then
        return SP_EL0;
    else
        case PSTATE.EL of
            when EL0  return SP_EL0;
            when EL1  return SP_EL1;
            when EL2  return SP_EL2;
            when EL3  return SP_EL3;
```

Library pseudocode for aarch64/functions/registers/SPMCFGR_EL1

```
// SPMCFGR_EL1[] - non-assignment form
// =====================================
// Read the current configuration of System Performance monitor for
// System PMU 's'.

bits(64) SPMCFGR_EL1[integer s];
```

Library pseudocode for aarch64/functions/registers/SPMCGCR_EL1

```
// SPMCGCR_EL1[] - non-assignment form
// =====================================
// Read the counter group configuration of System Performance monitor
// 's'.

bits(64) SPMCGCR_EL1[integer s];
```

Library pseudocode for aarch64/functions/registers/SPMCNTENCLR_EL0

```
// SPMCNTENCLR_EL0[] - non-assignment form
// =========================================
// Read the current mapping of disabled event counters for an 's'.

bits(64) SPMCNTENCLR_EL0[integer s];

// SPMCNTENCLR_EL0[] - assignment form
// ===================================
// Disable event counters for System PMU 's'.

SPMCNTENCLR_EL0[integer s] = bits(64) value;
```

Library pseudocode for aarch64/functions/registers/SPMCNTENSET_EL0

```
// SPMCNTENSET_EL0[] - non-assignment form
// =======================================
// Read the current mapping for enabled event counters of System PMU 's'.

bits(64) SPMCNTENSET_EL0[integer s];

// SPMCNTENSET_EL0[] - assignment form
// ===================================
// Enable event counters of System PMU 's'.

SPMCNTENSET_EL0[integer s] = bits(64) value;
```

Library pseudocode for aarch64/functions/registers/SPMCR_EL0

```
// SPMCR_EL0[] - non-assignment form
// =================================
// Read the control register for System PMU 's'.

bits(64) SPMCR_EL0[integer s];

// SPMCR_EL0[] - assignment form
// =============================
// Write to the control register for System PMU 's'.

SPMCR_EL0[integer s] = bits(64) value;
```

Library pseudocode for aarch64/functions/registers/SPMDEVAFF_EL1

```
// SPMDEVAFF_EL1[] - non-assignment form
// =====================================
// Read the discovery information for System PMU 's'.

bits(64) SPMDEVAFF_EL1[integer s];
```

Library pseudocode for aarch64/functions/registers/SPMDEVARCH_EL1

```
// SPMDEVARCH_EL1[] - non-assignment form
// ======================================
// Read the discovery information for System PMU 's'.

bits(64) SPMDEVARCH_EL1[integer s];
```

Library pseudocode for aarch64/functions/registers/SPMEVCNTR_EL0

```
// SPMEVCNTR_EL0[] - non-assignment form
// =====================================
// Read a System PMU Event Counter register for counter 'n' of a given
// System PMU 's'.

bits(64) SPMEVCNTR_EL0[integer s, integer n];

// SPMEVCNTR_EL0[] - assignment form
// =================================
// Write to a System PMU Event Counter register for counter 'n' of a given
// System PMU 's'.

SPMEVCNTR_EL0[integer s, integer n] = bits(64) value;
```

Library pseudocode for aarch64/functions/registers/SPMEVFILT2R_EL0

```
// SPMEVFILT2R_EL0[] - non-assignment form
// ==========================================
// Read the additional event selection controls for
// counter 'n' of a given System PMU 's'.

bits(64) SPMEVFILT2R_EL0[integer s, integer n];

// SPMEVFILT2R_EL0[] - assignment form
// ======================================
// Configure the additional event selection controls for
// counter 'n' of a given System PMU 's'.

SPMEVFILT2R_EL0[integer s, integer n] = bits(64) value;
```

Library pseudocode for aarch64/functions/registers/SPMEVFILTR_EL0

```
// SPMEVFILTR_EL0[] - non-assignment form
// =========================================
// Read the additional event selection controls for
// counter 'n' of a given System PMU 's'.

bits(64) SPMEVFILTR_EL0[integer s, integer n];

// SPMEVFILTR_EL0[] - assignment form
// =====================================
// Configure the additional event selection controls for
// counter 'n' of a given System PMU 's'.

SPMEVFILTR_EL0[integer s, integer n] = bits(64) value;
```

Library pseudocode for aarch64/functions/registers/SPMEVTYPER_EL0

```
// SPMEVTYPER_EL0[] - non-assignment form
// =========================================
// Read the current mapping of event with event counter SPMEVCNTR_EL0
// for counter 'n' of a given System PMU 's'.

bits(64) SPMEVTYPER_EL0[integer s, integer n];

// SPMEVTYPER_EL0[] - assignment form
// =====================================
// Configure which event increments the event counter SPMEVCNTR_EL0, for
// counter 'n' of a given System PMU 's'.

SPMEVTYPER_EL0[integer s, integer n] = bits(64) value;
```

Library pseudocode for aarch64/functions/registers/SPMIIDR_EL1

```
// SPMIIDR_EL1[] - non-assignment form
// ======================================
// Read the discovery information for System PMU 's'.

bits(64) SPMIIDR_EL1[integer s];
```

Library pseudocode for aarch64/functions/registers/SPMINTENCLR_EL1

```
// SPMINTENCLR_EL1[] - non-assignment form
// ==========================================
// Read the masking information for interrupt requests on overflows of
// implemented counters of System PMU 's'.

bits(64) SPMINTENCLR_EL1[integer s];

// SPMINTENCLR_EL1[] - assignment form
// ======================================
// Disable the generation of interrupt requests on overflows of
// implemented counters of System PMU 's'.

SPMINTENCLR_EL1[integer s] = bits(64) value;
```

Library pseudocode for aarch64/functions/registers/SPMINTENSET_EL1

```
// SPMINTENSET_EL1[] - non-assignment form
// ==========================================
// Read the masking information for interrupt requests on overflows of
// implemented counters of System PMU 's'.

bits(64) SPMINTENSET_EL1[integer s];

// SPMINTENSET_EL1[] - assignment form
// ======================================
// Disable the generation of interrupt requests on overflows of
// implemented counters for System PMU 's'.

SPMINTENSET_EL1[integer s] = bits(64) value;
```

Library pseudocode for aarch64/functions/registers/SPMOVSCLR_EL0

```
// SPMOVSCLR_EL0[] - non-assignment form
// ========================================
// Read the overflow bit clear status of implemented counters for System PMU 's'.

bits(64) SPMOVSCLR_EL0[integer s];

// SPMOVSCLR_EL0[] - assignment form
// ====================================
// Clear the overflow bit clear status of implemented counters for
// System PMU 's'.

SPMOVSCLR_EL0[integer s] = bits(64) value;
```

Library pseudocode for aarch64/functions/registers/SPMOVSSET_EL0

```
// SPMOVSSET_EL0[] - non-assignment form
// ========================================
// Read state of the overflow bit for the implemented event counters
// of System PMU 's'.

bits(64) SPMOVSSET_EL0[integer s];

// SPMOVSSET_EL0[] - assignment form
// ====================================
// Sets the state of the overflow bit for the implemented event counters
// of System PMU 's'.

SPMOVSSET_EL0[integer s] = bits(64) value;
```

Library pseudocode for aarch64/functions/registers/SPMROOTCR_EL3

```
// SPMROOTCR_EL3[] - non-assignment form
// =====================================
// Read the observability of Root and Realm events by System Performance
// Monitor for System PMU 's'.

bits(64) SPMROOTCR_EL3[integer s];

// SPMROOTCR_EL3[] - assignment form
// =================================
// Configure the observability of Root and Realm events by System
// Performance Monitor for System PMU 's'.

SPMROOTCR_EL3[integer s] = bits(64) value;
```

Library pseudocode for aarch64/functions/registers/SPMSCR_EL1

```
// SPMSCR_EL1[] - non-assignment form
// ==================================
// Read the observability of Secure events by System Performance Monitor
// for System PMU 's'.

bits(64) SPMSCR_EL1[integer s];

// SPMSCR_EL1[] - assignment form
// ==============================
// Configure the observability of secure events by System Performance
// Monitor for System PMU 's'.

SPMSCR_EL1[integer s] = bits(64) value;
```

Library pseudocode for aarch64/functions/registers/V

```
// V[] - assignment form
// =====================
// Write to SIMD&FP register with implicit extension from
// 8, 16, 32, 64 or 128 bits.

V[integer n, integer width] = bits(width) value
    assert n >= 0 && n <= 31;
    assert width IN {8,16,32,64,128};
    integer vlen = if IsSVEEnabled(PSTATE.EL) then CurrentVL else 128;
    if ConstrainUnpredictableBool(Unpredictable_SVEZEROUPPER) then
        _Z[n] = ZeroExtend(value, MAX_VL);
    else
        _Z[n]<vlen-1:0> = ZeroExtend(value, vlen);

// V[] - non-assignment form
// =========================
// Read from SIMD&FP register with implicit slice of 8, 16
// 32, 64 or 128 bits.

bits(width) V[integer n, integer width]
    assert n >= 0 && n <= 31;
    assert width IN {8,16,32,64,128};
    return _Z[n]<width-1:0>;
```

Library pseudocode for aarch64/functions/registers/Vpart

```
// Vpart[] - non-assignment form
// =============================
// Reads a 128-bit SIMD&FP register in up to two parts:
//  part 0 returns the bottom 8, 16, 32 or 64 bits of a value held in the register;
//  part 1 returns the top half of the bottom 64 bits or the top half of the 128-bit
//  value held in the register.

bits(width) Vpart[integer n, integer part, integer width]
    assert n >= 0 && n <= 31;
    assert part IN {0, 1};
    if part == 0 then
        assert width < 128;
        return V[n, width];
    else
        assert width IN {32,64};
        bits(128) vreg = V[n, 128];
        return vreg<(width * 2)-1:width>;

// Vpart[] - assignment form
// =========================
// Writes a 128-bit SIMD&FP register in up to two parts:
//  part 0 zero extends a 8, 16, 32, or 64-bit value to fill the whole register;
//  part 1 inserts a 64-bit value into the top half of the register.

Vpart[integer n, integer part, integer width] = bits(width) value
    assert n >= 0 && n <= 31;
    assert part IN {0, 1};
    if part == 0 then
        assert width < 128;
        V[n, width] = value;
    else
        assert width == 64;
        bits(64) vreg = V[n, 64];
        V[n, 128] = value<63:0> : vreg;
```

Library pseudocode for aarch64/functions/registers/X

```
// X[] - assignment form
// =====================
// Write to general-purpose register from either a 32-bit or a 64-bit value,
// where the size of the value is passed as an argument.

X[integer n, integer width] = bits(width) value
    assert n >= 0 && n <= 31;
    assert width IN {32,64};
    if n != 31 then
        _R[n] = ZeroExtend(value, 64);
    return;

// X[] - non-assignment form
// =========================
// Read from general-purpose register with an explicit slice of 8, 16, 32 or 64 bits.

bits(width) X[integer n, integer width]
    assert n >= 0 && n <= 31;
    assert width IN {8,16,32,64};
    if n != 31 then
        return _R[n]<width-1:0>;
    else
        return Zeros(width);
```

Library pseudocode for aarch64/functions/shiftreg/DecodeShift

```
// DecodeShift()
// =============
// Decode shift encodings

ShiftType DecodeShift(bits(2) op)
    case op of
        when '00'  return ShiftType_LSL;
        when '01'  return ShiftType_LSR;
        when '10'  return ShiftType_ASR;
        when '11'  return ShiftType_ROR;
```

Library pseudocode for aarch64/functions/shiftreg/ShiftReg

```
// ShiftReg()
// ==========
// Perform shift of a register operand

bits(N) ShiftReg(integer reg, ShiftType shiftype, integer amount, integer N)
    bits(N) result = X[reg, N];
    case shiftype of
        when ShiftType_LSL result = LSL(result, amount);
        when ShiftType_LSR result = LSR(result, amount);
        when ShiftType_ASR result = ASR(result, amount);
        when ShiftType_ROR result = ROR(result, amount);
    return result;
```

Library pseudocode for aarch64/functions/shiftreg/ShiftType

```
// ShiftType
// =========
// AArch64 register shifts.

enumeration ShiftType   {ShiftType_LSL, ShiftType_LSR, ShiftType_ASR, ShiftType_ROR};
```

Library pseudocode for aarch64/functions/sme/CounterToPredicate

```
// CounterToPredicate()
// ====================

bits(width) CounterToPredicate(bits(16) pred, integer width)
    integer count;
    integer esize;
    integer elements;
    constant integer VL = CurrentVL;
    constant integer PL = VL DIV 8;
    integer maxbit = HighestSetBit(CeilPow2(PL * 4)<15:0>);
    assert maxbit <= 14;
    bits(PL*4) result;
    boolean invert = pred<15> == '1';

    assert width == PL || width == PL*2 || width == PL*3 || width == PL*4;

    if IsZero(pred<3:0>) then
        return Zeros(width);

    case pred<3:0> of
        when 'xxx1'
            count = UInt(pred<maxbit:1>);
            esize = 8;
        when 'xx10'
            count = UInt(pred<maxbit:2>);
            esize = 16;
        when 'x100'
            count = UInt(pred<maxbit:3>);
            esize = 32;
        when '1000'
            count = UInt(pred<maxbit:4>);
            esize = 64;

    elements = (VL * 4) DIV esize;
    result = Zeros(PL*4);
    constant integer psize = esize DIV 8;
    for e = 0 to elements-1
        bit pbit = if e < count then '1' else '0';
        if invert then
            pbit = NOT(pbit);
        Elem[result, e, psize] = ZeroExtend(pbit, psize);

    return result<width-1:0>;
```

Library pseudocode for aarch64/functions/sme/EncodePredCount

```
// EncodePredCount()
// =================

bits(width) EncodePredCount(integer esize, integer elements,
                            integer count_in, boolean invert_in, integer width)
    integer count = count_in;
    boolean invert = invert_in;
    constant integer PL = CurrentVL DIV 8;
    assert width == PL;
    assert esize IN {8, 16, 32, 64};
    assert count >=0 && count <= elements;
    bits(16) pred;

    if count == 0 then
        return Zeros(width);

    if invert then
        count = elements - count;
    elsif count == elements then
        count = 0;
        invert = TRUE;

    bit inv = (if invert then '1' else '0');
    case esize of
        when 8  pred = inv : count<13:0> :    '1';
        when 16 pred = inv : count<12:0> :   '10';
        when 32 pred = inv : count<11:0> :  '100';
        when 64 pred = inv : count<10:0> : '1000';

    return ZeroExtend(pred, width);
```

Library pseudocode for aarch64/functions/sme/HaveSME

```
// HaveSME()
// =========
// Returns TRUE if the SME extension is implemented, FALSE otherwise.

boolean HaveSME()
    return IsFeatureImplemented(FEAT_SME);
```

Library pseudocode for aarch64/functions/sme/HaveSME2

```
// HaveSME2()
// ==========
// Returns TRUE if the SME2 extension is implemented, FALSE otherwise.

boolean HaveSME2()
    return IsFeatureImplemented(FEAT_SME2);
```

Library pseudocode for aarch64/functions/sme/HaveSME2p1

```
// HaveSME2p1()
// ============
// Returns TRUE if the SME2.1 extension is implemented, FALSE otherwise.

boolean HaveSME2p1()
    return IsFeatureImplemented(FEAT_SME2p1);
```

Library pseudocode for aarch64/functions/sme/HaveSMEB16B16

```
// HaveSMEB16B16()
// ===============
// Returns TRUE if the SME2.1 non-widening BFloat16 instructions are implemented, FALSE otherwise.

boolean HaveSMEB16B16()
    return IsFeatureImplemented(FEAT_SVE_B16B16);
```

Library pseudocode for aarch64/functions/sme/HaveSMEF16F16

```
// HaveSMEF16F16()
// ===============
// Returns TRUE if the SME2.1 half-precision instructions are implemented, FALSE otherwise.

boolean HaveSMEF16F16()
    return IsFeatureImplemented(FEAT_SME_F16F16);
```

Library pseudocode for aarch64/functions/sme/HaveSMEF64F64

```
// HaveSMEF64F64()
// ===============
// Returns TRUE if the SMEF64F64 extension is implemented, FALSE otherwise.

boolean HaveSMEF64F64()
    return IsFeatureImplemented(FEAT_SME_F64F64);
```

Library pseudocode for aarch64/functions/sme/HaveSMEI16I64

```
// HaveSMEI16I64()
// ===============
// Returns TRUE if the SMEI16I64 extension is implemented, FALSE otherwise.

boolean HaveSMEI16I64()
    return IsFeatureImplemented(FEAT_SME_I16I64);
```

Library pseudocode for aarch64/functions/sme/Lookup

```
bits(512) _ZT0;
```

Library pseudocode for aarch64/functions/sme/PredCountTest

```
// PredCountTest()
// ===============

bits(4) PredCountTest(integer elements, integer count, boolean invert)
    bit n, z, c, v;
    z = (if count == 0 then '1' else '0');              // none active
    if !invert then
        n = (if count != 0 then '1' else '0');          // first active
        c = (if count == elements then '0' else '1');   // NOT last active
    else
        n = (if count == elements then '1' else '0');   // first active
        c = (if count != 0 then '0' else '1');          // NOT last active
    v = '0';

    return n:z:c:v;
```

Library pseudocode for aarch64/functions/sme/System

```
// System Registers
// ================

array bits(MAX_VL) _ZA[0..255];
```

Library pseudocode for aarch64/functions/sme/ZAhslice

```
// ZAhslice[] - non-assignment form
// ================================

bits(width) ZAhslice[integer tile, integer esize, integer slice, integer width]
    assert esize IN {8, 16, 32, 64, 128};
    integer tiles = esize DIV 8;
    assert tile >= 0 && tile < tiles;
    integer slices = CurrentSVL DIV esize;
    assert slice >= 0 && slice < slices;

    return ZAvector[tile + slice * tiles, width];

// ZAhslice[] - assignment form
// ============================

ZAhslice[integer tile, integer esize, integer slice, integer width] = bits(width) value
    assert esize IN {8, 16, 32, 64, 128};
    integer tiles = esize DIV 8;
    assert tile >= 0 && tile < tiles;
    integer slices = CurrentSVL DIV esize;
    assert slice >= 0 && slice < slices;

    ZAvector[tile + slice * tiles, width] = value;
```

Library pseudocode for aarch64/functions/sme/ZAslice

```
// ZAslice[] - non-assignment form
// ===============================

bits(width) ZAslice[integer tile, integer esize, boolean vertical, integer slice, integer width]
    bits(width) result;

    if vertical then
        result = ZAvslice[tile, esize, slice, width];
    else
        result = ZAhslice[tile, esize, slice, width];

    return result;

// ZAslice[] - assignment form
// ===========================

ZAslice[integer tile, integer esize, boolean vertical,
        integer slice, integer width] = bits(width) value
    if vertical then
        ZAvslice[tile, esize, slice, width] = value;
    else
        ZAhslice[tile, esize, slice, width] = value;
```

Library pseudocode for aarch64/functions/sme/ZAtile

```
// ZAtile[] - non-assignment form
// ==============================

bits(width) ZAtile[integer tile, integer esize, integer width]
    constant integer SVL = CurrentSVL;
    integer slices = SVL DIV esize;
    assert width == SVL * slices;
    bits(width) result;

    for slice = 0 to slices-1
        Elem[result, slice, SVL] = ZAhslice[tile, esize, slice, SVL];

    return result;

// ZAtile[] - assignment form
// ==========================

ZAtile[integer tile, integer esize, integer width] = bits(width) value
    constant integer SVL = CurrentSVL;
    integer slices = SVL DIV esize;
    assert width == SVL * slices;

    for slice = 0 to slices-1
        ZAhslice[tile, esize, slice, SVL] = Elem[value, slice, SVL];
```

Library pseudocode for aarch64/functions/sme/ZAvector

```
// ZAvector[] - non-assignment form
// ================================

bits(width) ZAvector[integer index, integer width]
    assert width == CurrentSVL;
    assert index >= 0 && index < (width DIV 8);

    return _ZA[index]<width-1:0>;

// ZAvector[] - assignment form
// ============================

ZAvector[integer index, integer width] = bits(width) value
    assert width == CurrentSVL;
    assert index >= 0 && index < (width DIV 8);

    if ConstrainUnpredictableBool(Unpredictable_SMEZEROUPPER) then
        _ZA[index] = ZeroExtend(value, MAX_VL);
    else
        _ZA[index]<width-1:0> = value;
```

Library pseudocode for aarch64/functions/sme/ZAvslice

```
// ZAvslice[] - non-assignment form
// ================================

bits(width) ZAvslice[integer tile, integer esize, integer slice, integer width]
    integer slices = CurrentSVL DIV esize;
    bits(width) result;

    for s = 0 to slices-1
        bits(width) hslice = ZAhslice[tile, esize, s, width];
        Elem[result, s, esize] = Elem[hslice, slice, esize];

    return result;

// ZAvslice[] - assignment form
// ============================

ZAvslice[integer tile, integer esize, integer slice, integer width] = bits(width) value
    integer slices = CurrentSVL DIV esize;

    for s = 0 to slices-1
        bits(width) hslice = ZAhslice[tile, esize, s, width];
        Elem[hslice, slice, esize] = Elem[value, s, esize];
        ZAhslice[tile, esize, s, width] = hslice;
```

Library pseudocode for aarch64/functions/sme/ZT0

```
// ZT0[] - non-assignment form
// ===========================

bits(width) ZT0[integer width]
    assert width == 512;
    return _ZT0<width-1:0>;

// ZT0[] - assignment form
// =======================

ZT0[integer width] = bits(width) value
    assert width == 512;
    _ZT0<width-1:0> = value;
```

Library pseudocode for aarch64/functions/sve/AArch32.IsFPEnabled

```
// AArch32.IsFPEnabled()
// =====================
// Returns TRUE if access to the SIMD&FP instructions or System registers are
// enabled at the target exception level in AArch32 state and FALSE otherwise.

boolean AArch32.IsFPEnabled(bits(2) el)
    if el == EL0 && !ELUsingAArch32(EL1) then
        return AArch64.IsFPEnabled(el);

    if HaveEL(EL3) && ELUsingAArch32(EL3) && CurrentSecurityState() == SS_NonSecure then
        // Check if access disabled in NSACR
        if NSACR.cp10 == '0' then return FALSE;

    if el IN {EL0, EL1} then
        // Check if access disabled in CPACR
        boolean disabled;
        case CPACR.cp10 of
            when '00' disabled = TRUE;
            when '01' disabled = el == EL0;
            when '10' disabled = ConstrainUnpredictableBool(Unpredictable_RESCPACR);
            when '11' disabled = FALSE;
        if disabled then return FALSE;

    if el IN {EL0, EL1, EL2} && EL2Enabled() then
        if !ELUsingAArch32(EL2) then
            return AArch64.IsFPEnabled(EL2);
        if HCPTR.TCP10 == '1' then return FALSE;

    if HaveEL(EL3) && !ELUsingAArch32(EL3) then
        // Check if access disabled in CPTR_EL3
        if CPTR_EL3.TFP == '1' then return FALSE;

    return TRUE;
```

Library pseudocode for aarch64/functions/sve/AArch64.IsFPEnabled

```
// AArch64.IsFPEnabled()
// =====================
// Returns TRUE if access to the SIMD&FP instructions or System registers are
// enabled at the target exception level in AArch64 state and FALSE otherwise.

boolean AArch64.IsFPEnabled(bits(2) el)
    // Check if access disabled in CPACR_EL1
    if el IN {EL0, EL1} && !IsInHost() then
        // Check SIMD&FP at EL0/EL1
        boolean disabled;
        case CPACR_EL1.FPEN of
            when 'x0' disabled = TRUE;
            when '01' disabled = el == EL0;
            when '11' disabled = FALSE;
        if disabled then return FALSE;

    // Check if access disabled in CPTR_EL2
    if el IN {EL0, EL1, EL2} && EL2Enabled() then
        if HaveVirtHostExt() && HCR_EL2.E2H == '1' then
            boolean disabled;
            case CPTR_EL2.FPEN of
                when 'x0' disabled = TRUE;
                when '01' disabled = el == EL0 && HCR_EL2.TGE == '1';
                when '11' disabled = FALSE;
            if disabled then return FALSE;
        else
            if CPTR_EL2.TFP == '1' then return FALSE;

    // Check if access disabled in CPTR_EL3
    if HaveEL(EL3) then
        if CPTR_EL3.TFP == '1' then return FALSE;

    return TRUE;
```

Library pseudocode for aarch64/functions/sve/ActivePredicateElement

```
// ActivePredicateElement()
// ========================
// Returns TRUE if the predicate bit is 1 and FALSE otherwise

boolean ActivePredicateElement(bits(N) pred, integer e, integer esize)
    assert esize IN {8, 16, 32, 64, 128};
    integer n = e * (esize DIV 8);
    assert n >= 0 && n < N;
    return pred<n> == '1';
```

Library pseudocode for aarch64/functions/sve/AnyActiveElement

```
// AnyActiveElement()
// ==================
// Return TRUE if there is at least one active element in mask. Otherwise,
// return FALSE.

boolean AnyActiveElement(bits(N) mask, integer esize)
    return LastActiveElement(mask, esize) >= 0;
```

Library pseudocode for aarch64/functions/sve/BitDeposit

```
// BitDeposit()
// ============
// Deposit the least significant bits from DATA into result positions
// selected by nonzero bits in MASK, setting other result bits to zero.

bits(N) BitDeposit (bits(N) data, bits(N) mask)
    bits(N) res = Zeros(N);
    integer db = 0;
    for rb = 0 to N-1
        if mask<rb> == '1' then
            res<rb> = data<db>;
            db = db + 1;
    return res;
```

Library pseudocode for aarch64/functions/sve/BitExtract

```
// BitExtract()
// ============
// Extract and pack DATA bits selected by the nonzero bits in MASK into
// the least significant result bits, setting other result bits to zero.

bits(N) BitExtract (bits(N) data, bits(N) mask)
    bits(N) res = Zeros(N);
    integer rb = 0;
    for db = 0 to N-1
        if mask<db> == '1' then
            res<rb> = data<db>;
            rb = rb + 1;
    return res;
```

Library pseudocode for aarch64/functions/sve/BitGroup

```
// BitGroup()
// ==========
// Extract and pack DATA bits selected by the nonzero bits in MASK into
// the least significant result bits, and pack unselected bits into the
// most significant result bits.

bits(N) BitGroup (bits(N) data, bits(N) mask)
    bits(N) res;
    integer rb = 0;

    // compress masked bits to right
    for db = 0 to N-1
        if mask<db> == '1' then
            res<rb> = data<db>;
            rb = rb + 1;
    // compress unmasked bits to left
    for db = 0 to N-1
        if mask<db> == '0' then
            res<rb> = data<db>;
            rb = rb + 1;
    return res;
```

Library pseudocode for aarch64/functions/sve/CeilPow2

```
// CeilPow2()
// ==========
// For a positive integer X, return the smallest power of 2 >= X

integer CeilPow2(integer x)
    if x == 0 then return 0;
    if x == 1 then return 2;
    return FloorPow2(x - 1) * 2;
```

Library pseudocode for aarch64/functions/sve/CheckNonStreamingSVEEnabled

```
// CheckNonStreamingSVEEnabled()
// =============================
// Checks for traps on SVE instructions that are not legal in streaming mode.

CheckNonStreamingSVEEnabled()
    CheckSVEEnabled();

    if HaveSME() && PSTATE.SM == '1' && !IsFullA64Enabled() then
        SMEAccessTrap(SMEExceptionType_Streaming, PSTATE.EL);
```

Library pseudocode for aarch64/functions/sve/CheckOriginalSVEEnabled

```
// CheckOriginalSVEEnabled()
// =========================
// Checks for traps on SVE instructions and instructions that access SVE System
// registers.

CheckOriginalSVEEnabled()
    assert HaveSVE();
    boolean disabled;

    if (HaveEL(EL3) && (CPTR_EL3.EZ == '0' || CPTR_EL3.TFP == '1') &&
        EL3SDDUndefPriority()) then
        UNDEFINED;

    // Check if access disabled in CPACR_EL1
    if PSTATE.EL IN {EL0, EL1} && !IsInHost() then
        // Check SVE at EL0/EL1
        case CPACR_EL1.ZEN of
            when 'x0' disabled = TRUE;
            when '01' disabled = PSTATE.EL == EL0;
            when '11' disabled = FALSE;
        if disabled then SVEAccessTrap(EL1);

        // Check SIMD&FP at EL0/EL1
        case CPACR_EL1.FPEN of
            when 'x0' disabled = TRUE;
            when '01' disabled = PSTATE.EL == EL0;
            when '11' disabled = FALSE;
        if disabled then AArch64.AdvSIMDFPAccessTrap(EL1);

    // Check if access disabled in CPTR_EL2
    if PSTATE.EL IN {EL0, EL1, EL2} && EL2Enabled() then
        if HaveVirtHostExt() && HCR_EL2.E2H == '1' then
            // Check SVE at EL2
            case CPTR_EL2.ZEN of
                when 'x0' disabled = TRUE;
                when '01' disabled = PSTATE.EL == EL0 && HCR_EL2.TGE == '1';
                when '11' disabled = FALSE;
            if disabled then SVEAccessTrap(EL2);

            // Check SIMD&FP at EL2
            case CPTR_EL2.FPEN of
                when 'x0' disabled = TRUE;
                when '01' disabled = PSTATE.EL == EL0 && HCR_EL2.TGE == '1';
                when '11' disabled = FALSE;
            if disabled then AArch64.AdvSIMDFPAccessTrap(EL2);
        else
            if CPTR_EL2.TZ == '1' then SVEAccessTrap(EL2);
            if CPTR_EL2.TFP == '1' then AArch64.AdvSIMDFPAccessTrap(EL2);

    // Check if access disabled in CPTR_EL3
    if HaveEL(EL3) then
        if CPTR_EL3.EZ == '0' then
            if EL3SDDUndef() then
                UNDEFINED;
            else
                SVEAccessTrap(EL3);

        if CPTR_EL3.TFP == '1' then
            if EL3SDDUndef() then
                UNDEFINED;
            else
                AArch64.AdvSIMDFPAccessTrap(EL3);
```

Library pseudocode for aarch64/functions/sve/CheckSMEAccess

```
// CheckSMEAccess()
// ================
// Check that access to SME System registers is enabled.

CheckSMEAccess()
    boolean disabled;
    // Check if access disabled in CPACR_EL1
    if PSTATE.EL IN {EL0, EL1} && !IsInHost() then
        // Check SME at EL0/EL1
        case CPACR_EL1.SMEN of
            when 'x0' disabled = TRUE;
            when '01' disabled = PSTATE.EL == EL0;
            when '11' disabled = FALSE;
        if disabled then SMEAccessTrap(SMEExceptionType_AccessTrap, EL1);

    if PSTATE.EL IN {EL0, EL1, EL2} && EL2Enabled() then
        if HaveVirtHostExt() && HCR_EL2.E2H == '1' then
            // Check SME at EL2
            case CPTR_EL2.SMEN of
                when 'x0' disabled = TRUE;
                when '01' disabled = PSTATE.EL == EL0 && HCR_EL2.TGE == '1';
                when '11' disabled = FALSE;
            if disabled then SMEAccessTrap(SMEExceptionType_AccessTrap, EL2);
        else
            if CPTR_EL2.TSM == '1' then SMEAccessTrap(SMEExceptionType_AccessTrap, EL2);

    // Check if access disabled in CPTR_EL3
    if HaveEL(EL3) then
        if CPTR_EL3.ESM == '0' then SMEAccessTrap(SMEExceptionType_AccessTrap, EL3);
```

Library pseudocode for aarch64/functions/sve/CheckSMEAndZAEnabled

```
// CheckSMEAndZAEnabled()
// ======================

CheckSMEAndZAEnabled()
    CheckSMEEnabled();

    if PSTATE.ZA == '0' then
        SMEAccessTrap(SMEExceptionType_InactiveZA, PSTATE.EL);
```

Library pseudocode for aarch64/functions/sve/CheckSMEEnabled

```
// CheckSMEEnabled()
// =================

CheckSMEEnabled()
    boolean disabled;
    // Check if access disabled in CPACR_EL1
    if PSTATE.EL IN {EL0, EL1} && !IsInHost() then
        // Check SME at EL0/EL1
        case CPACR_EL1.SMEN of
            when 'x0' disabled = TRUE;
            when '01' disabled = PSTATE.EL == EL0;
            when '11' disabled = FALSE;
        if disabled then SMEAccessTrap(SMEExceptionType_AccessTrap, EL1);

        // Check SIMD&FP at EL0/EL1
        case CPACR_EL1.FPEN of
            when 'x0' disabled = TRUE;
            when '01' disabled = PSTATE.EL == EL0;
            when '11' disabled = FALSE;
        if disabled then AArch64.AdvSIMDFPAccessTrap(EL1);

    if PSTATE.EL IN {EL0, EL1, EL2} && EL2Enabled() then
        if HaveVirtHostExt() && HCR_EL2.E2H == '1' then
            // Check SME at EL2
            case CPTR_EL2.SMEN of
                when 'x0' disabled = TRUE;
                when '01' disabled = PSTATE.EL == EL0 && HCR_EL2.TGE == '1';
                when '11' disabled = FALSE;
            if disabled then SMEAccessTrap(SMEExceptionType_AccessTrap, EL2);

            // Check SIMD&FP at EL2
            case CPTR_EL2.FPEN of
                when 'x0' disabled = TRUE;
                when '01' disabled = PSTATE.EL == EL0 && HCR_EL2.TGE == '1';
                when '11' disabled = FALSE;
            if disabled then AArch64.AdvSIMDFPAccessTrap(EL2);
        else
            if CPTR_EL2.TSM == '1' then SMEAccessTrap(SMEExceptionType_AccessTrap, EL2);
            if CPTR_EL2.TFP == '1' then AArch64.AdvSIMDFPAccessTrap(EL2);

    // Check if access disabled in CPTR_EL3
    if HaveEL(EL3) then
        if CPTR_EL3.ESM == '0' then SMEAccessTrap(SMEExceptionType_AccessTrap, EL3);
        if CPTR_EL3.TFP == '1' then AArch64.AdvSIMDFPAccessTrap(EL3);
```

Library pseudocode for aarch64/functions/sve/CheckSMEZT0Enabled

```
// CheckSMEZT0Enabled()
// ====================
// Checks for ZT0 enabled.

CheckSMEZT0Enabled()
    // Check if ZA and ZT0 are inactive in PSTATE
    if PSTATE.ZA == '0' then
        SMEAccessTrap(SMEExceptionType_InactiveZA, PSTATE.EL);

    // Check if EL0/EL1 accesses to ZT0 are disabled in SMCR_EL1
    if PSTATE.EL IN {EL0, EL1} && !IsInHost() then
        if SMCR_EL1.EZT0 == '0' then
            SMEAccessTrap(SMEExceptionType_InaccessibleZT0, EL1);

    // Check if EL0/EL1/EL2 accesses to ZT0 are disabled in SMCR_EL2
    if PSTATE.EL IN {EL0, EL1, EL2} && EL2Enabled() then
        if SMCR_EL2.EZT0 == '0' then
            SMEAccessTrap(SMEExceptionType_InaccessibleZT0, EL2);

    // Check if all accesses to ZT0 are disabled in SMCR_EL3
    if HaveEL(EL3) then
        if SMCR_EL3.EZT0 == '0' then
            SMEAccessTrap(SMEExceptionType_InaccessibleZT0, EL3);
```

Library pseudocode for aarch64/functions/sve/CheckSVEEnabled

```
// CheckSVEEnabled()
// =================
// Checks for traps on SVE instructions and instructions that
// access SVE System registers.

CheckSVEEnabled()
    if HaveSME() && PSTATE.SM == '1' then
        CheckSMEEnabled();
    elsif HaveSME() && !HaveSVE() then
        CheckStreamingSVEEnabled();
    else
        CheckOriginalSVEEnabled();
```

Library pseudocode for aarch64/functions/sve/CheckStreamingSVEAndZAEnabled

```
// CheckStreamingSVEAndZAEnabled()
// ===============================

CheckStreamingSVEAndZAEnabled()
    CheckStreamingSVEEnabled();

    if PSTATE.ZA == '0' then
        SMEAccessTrap(SMEExceptionType_InactiveZA, PSTATE.EL);
```

Library pseudocode for aarch64/functions/sve/CheckStreamingSVEEnabled

```
// CheckStreamingSVEEnabled()
// ==========================

CheckStreamingSVEEnabled()
    CheckSMEEnabled();

    if PSTATE.SM == '0' then
        SMEAccessTrap(SMEExceptionType_NotStreaming, PSTATE.EL);
```

Library pseudocode for aarch64/functions/sve/CurrentNSVL

```
// CurrentNSVL - non-assignment form
// =================================
// Non-Streaming VL

integer CurrentNSVL
    integer vl;

    if PSTATE.EL == EL1 || (PSTATE.EL == EL0 && !IsInHost()) then
        vl = UInt(ZCR_EL1.LEN);

    if PSTATE.EL == EL2 || (PSTATE.EL == EL0 && IsInHost()) then
        vl = UInt(ZCR_EL2.LEN);
    elsif PSTATE.EL IN {EL0, EL1} && EL2Enabled() then
        vl = Min(vl, UInt(ZCR_EL2.LEN));

    if PSTATE.EL == EL3 then
        vl = UInt(ZCR_EL3.LEN);
    elsif HaveEL(EL3) then
        vl = Min(vl, UInt(ZCR_EL3.LEN));

    vl = (vl + 1) * 128;
    vl = ImplementedSVEVectorLength(vl);

    return vl;
```

Library pseudocode for aarch64/functions/sve/CurrentSVL

```
// CurrentSVL - non-assignment form
// ================================
// Streaming SVL

integer CurrentSVL
    integer vl;

    if PSTATE.EL == EL1 || (PSTATE.EL == EL0 && !IsInHost()) then
        vl = UInt(SMCR_EL1.LEN);

    if PSTATE.EL == EL2 || (PSTATE.EL == EL0 && IsInHost()) then
        vl = UInt(SMCR_EL2.LEN);
    elsif PSTATE.EL IN {EL0, EL1} && EL2Enabled() then
        vl = Min(vl, UInt(SMCR_EL2.LEN));

    if PSTATE.EL == EL3 then
        vl = UInt(SMCR_EL3.LEN);
    elsif HaveEL(EL3) then
        vl = Min(vl, UInt(SMCR_EL3.LEN));

    vl = (vl + 1) * 128;
    vl = ImplementedSMEVectorLength(vl);

    return vl;
```

Library pseudocode for aarch64/functions/sve/CurrentVL

```
// CurrentVL - non-assignment form
// ===============================

integer CurrentVL
    return if HaveSME() && PSTATE.SM == '1' then CurrentSVL else CurrentNSVL;
```

Library pseudocode for aarch64/functions/sve/DecodePredCount

```
// DecodePredCount()
// =================

integer DecodePredCount(bits(5) pattern, integer esize)
    integer elements = CurrentVL DIV esize;
    integer numElem;
    case pattern of
        when '00000' numElem = FloorPow2(elements);
        when '00001' numElem = if elements >= 1 then 1 else 0;
        when '00010' numElem = if elements >= 2 then 2 else 0;
        when '00011' numElem = if elements >= 3 then 3 else 0;
        when '00100' numElem = if elements >= 4 then 4 else 0;
        when '00101' numElem = if elements >= 5 then 5 else 0;
        when '00110' numElem = if elements >= 6 then 6 else 0;
        when '00111' numElem = if elements >= 7 then 7 else 0;
        when '01000' numElem = if elements >= 8 then 8 else 0;
        when '01001' numElem = if elements >= 16 then 16 else 0;
        when '01010' numElem = if elements >= 32 then 32 else 0;
        when '01011' numElem = if elements >= 64 then 64 else 0;
        when '01100' numElem = if elements >= 128 then 128 else 0;
        when '01101' numElem = if elements >= 256 then 256 else 0;
        when '11101' numElem = elements - (elements MOD 4);
        when '11110' numElem = elements - (elements MOD 3);
        when '11111' numElem = elements;
        otherwise    numElem = 0;
    return numElem;
```

Library pseudocode for aarch64/functions/sve/ElemFFR

```
// ElemFFR[] - non-assignment form
// ===============================

bit ElemFFR[integer e, integer esize]
    return PredicateElement(_FFR, e, esize);

// ElemFFR[] - assignment form
// ===========================

ElemFFR[integer e, integer esize] = bit value
    integer psize = esize DIV 8;
    integer n = e * psize;
    assert n >= 0 && (n + psize) <= CurrentVL DIV 8;
    _FFR<(n+psize)-1:n> = ZeroExtend(value, psize);
    return;
```

Library pseudocode for aarch64/functions/sve/FFR

```
// FFR[] - non-assignment form
// ===========================

bits(width) FFR[integer width]
    assert width == CurrentVL DIV 8;
    return _FFR<width-1:0>;

// FFR[] - assignment form
// =======================

FFR[integer width] = bits(width) value
    assert width == CurrentVL DIV 8;
    if ConstrainUnpredictableBool(Unpredictable_SVEZEROUPPER) then
        _FFR = ZeroExtend(value, MAX_PL);
    else
        _FFR<width-1:0> = value;
```

Library pseudocode for aarch64/functions/sve/FPCompareNE

```
// FPCompareNE()
// =============

boolean FPCompareNE(bits(N) op1, bits(N) op2, FPCRType fpcr)
    assert N IN {16,32,64};
    boolean result;
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    op1_nan = type1 IN {FPType_SNaN, FPType_QNaN};
    op2_nan = type2 IN {FPType_SNaN, FPType_QNaN};

    if op1_nan || op2_nan then
        result = TRUE;
        if type1 == FPType_SNaN || type2 == FPType_SNaN then
            FPProcessException(FPExc_InvalidOp, fpcr);
    else // All non-NaN cases can be evaluated on the values produced by FPUnpack()
        result = (value1 != value2);
        FPProcessDenorms(type1, type2, N, fpcr);

    return result;
```

Library pseudocode for aarch64/functions/sve/FPCompareUN

```
// FPCompareUN()
// =============

boolean FPCompareUN(bits(N) op1, bits(N) op2, FPCRType fpcr)
    assert N IN {16,32,64};
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);

    if type1 == FPType_SNaN || type2 == FPType_SNaN then
        FPProcessException(FPExc_InvalidOp, fpcr);

    result = type1 IN {FPType_SNaN, FPType_QNaN} || type2 IN {FPType_SNaN, FPType_QNaN};
    if !result then
        FPProcessDenorms(type1, type2, N, fpcr);

    return result;
```

Library pseudocode for aarch64/functions/sve/FPConvertSVE

```
// FPConvertSVE()
// ==============

bits(M) FPConvertSVE(bits(N) op, FPCRType fpcr_in, FPRounding rounding, integer M)
    FPCRType fpcr = fpcr_in;
    fpcr.AHP = '0';
    return FPConvert(op, fpcr, rounding, M);

// FPConvertSVE()
// ==============

bits(M) FPConvertSVE(bits(N) op, FPCRType fpcr_in, integer M)
    FPCRType fpcr = fpcr_in;
    fpcr.AHP = '0';
    return FPConvert(op, fpcr, FPRoundingMode(fpcr), M);
```

Library pseudocode for aarch64/functions/sve/FPExpA

```
// FPExpA()
// ========

bits(N) FPExpA(bits(N) op)
    assert N IN {16,32,64};
    bits(N) result;
    bits(N) coeff;
    integer idx = if N == 16 then UInt(op<4:0>) else UInt(op<5:0>);
    coeff = FPExpCoefficient[idx, N];
    if N == 16 then
        result<15:0> = '0':op<9:5>:coeff<9:0>;
    elsif N == 32 then
        result<31:0> = '0':op<13:6>:coeff<22:0>;
    else // N == 64
        result<63:0> = '0':op<16:6>:coeff<51:0>;

    return result;
```

Library pseudocode for aarch64/functions/sve/FPExpCoefficient

```
// FPExpCoefficient()
// ==================

bits(N) FPExpCoefficient[integer index, integer N]
    assert N IN {16,32,64};
    integer result;

    if N == 16 then
        case index of
            when  0 result = 0x0000;
            when  1 result = 0x0016;
            when  2 result = 0x002d;
            when  3 result = 0x0045;
            when  4 result = 0x005d;
            when  5 result = 0x0075;
            when  6 result = 0x008e;
            when  7 result = 0x00a8;
            when  8 result = 0x00c2;
            when  9 result = 0x00dc;
            when 10 result = 0x00f8;
            when 11 result = 0x0114;
            when 12 result = 0x0130;
            when 13 result = 0x014d;
            when 14 result = 0x016b;
            when 15 result = 0x0189;
            when 16 result = 0x01a8;
            when 17 result = 0x01c8;
            when 18 result = 0x01e8;
            when 19 result = 0x0209;
            when 20 result = 0x022b;
            when 21 result = 0x024e;
            when 22 result = 0x0271;
            when 23 result = 0x0295;
            when 24 result = 0x02ba;
            when 25 result = 0x02e0;
            when 26 result = 0x0306;
            when 27 result = 0x032e;
            when 28 result = 0x0356;
            when 29 result = 0x037f;
            when 30 result = 0x03a9;
            when 31 result = 0x03d4;

    elsif N == 32 then
        case index of
            when  0 result = 0x000000;
            when  1 result = 0x0164d2;
            when  2 result = 0x02cd87;
            when  3 result = 0x043a29;
            when  4 result = 0x05aac3;
            when  5 result = 0x071f62;
            when  6 result = 0x08980f;
            when  7 result = 0x0a14d5;
            when  8 result = 0x0b95c2;
            when  9 result = 0x0d1adf;
            when 10 result = 0x0ea43a;
            when 11 result = 0x1031dc;
            when 12 result = 0x11c3d3;
            when 13 result = 0x135a2b;
            when 14 result = 0x14f4f0;
            when 15 result = 0x16942d;
            when 16 result = 0x1837f0;
            when 17 result = 0x19e046;
            when 18 result = 0x1b8d3a;
            when 19 result = 0x1d3eda;
            when 20 result = 0x1ef532;
            when 21 result = 0x20b051;
            when 22 result = 0x227043;
            when 23 result = 0x243516;
            when 24 result = 0x25fed7;
            when 25 result = 0x27cd94;
```

```
            when 26 result = 0x29a15b;
            when 27 result = 0x2b7a3a;
            when 28 result = 0x2d583f;
            when 29 result = 0x2f3b79;
            when 30 result = 0x3123f6;
            when 31 result = 0x3311c4;
            when 32 result = 0x3504f3;
            when 33 result = 0x36fd92;
            when 34 result = 0x38fbaf;
            when 35 result = 0x3aff5b;
            when 36 result = 0x3d08a4;
            when 37 result = 0x3f179a;
            when 38 result = 0x412c4d;
            when 39 result = 0x4346cd;
            when 40 result = 0x45672a;
            when 41 result = 0x478d75;
            when 42 result = 0x49b9be;
            when 43 result = 0x4bec15;
            when 44 result = 0x4e248c;
            when 45 result = 0x506334;
            when 46 result = 0x52a81e;
            when 47 result = 0x54f35b;
            when 48 result = 0x5744fd;
            when 49 result = 0x599d16;
            when 50 result = 0x5bfbb8;
            when 51 result = 0x5e60f5;
            when 52 result = 0x60ccdf;
            when 53 result = 0x633f89;
            when 54 result = 0x65b907;
            when 55 result = 0x68396a;
            when 56 result = 0x6ac0c7;
            when 57 result = 0x6d4f30;
            when 58 result = 0x6fe4ba;
            when 59 result = 0x728177;
            when 60 result = 0x75257d;
            when 61 result = 0x77d0df;
            when 62 result = 0x7a83b3;
            when 63 result = 0x7d3e0c;

    else // N == 64
        case index of
            when  0 result = 0x0000000000000;
            when  1 result = 0x02C9A3E778061;
            when  2 result = 0x059B0D3158574;
            when  3 result = 0x0874518759BC8;
            when  4 result = 0x0B5586CF9890F;
            when  5 result = 0x0E3EC32D3D1A2;
            when  6 result = 0x11301D0125B51;
            when  7 result = 0x1429AAEA92DE0;
            when  8 result = 0x172B83C7D517B;
            when  9 result = 0x1A35BEB6FCB75;
            when 10 result = 0x1D4873168B9AA;
            when 11 result = 0x2063B88628CD6;
            when 12 result = 0x2387A6E756238;
            when 13 result = 0x26B4565E27CDD;
            when 14 result = 0x29E9DF51FDEE1;
            when 15 result = 0x2D285A6E4030B;
            when 16 result = 0x306FE0A31B715;
            when 17 result = 0x33C08B26416FF;
            when 18 result = 0x371A7373AA9CB;
            when 19 result = 0x3A7DB34E59FF7;
            when 20 result = 0x3DEA64C123422;
            when 21 result = 0x4160A21F72E2A;
            when 22 result = 0x44E086061892D;
            when 23 result = 0x486A2B5C13CD0;
            when 24 result = 0x4BFDAD5362A27;
            when 25 result = 0x4F9B2769D2CA7;
            when 26 result = 0x5342B569D4F82;
            when 27 result = 0x56F4736B527DA;
            when 28 result = 0x5AB07DD485429;
```

```
            when 29 result = 0x5E76F15AD2148;
            when 30 result = 0x6247EB03A5585;
            when 31 result = 0x6623882552225;
            when 32 result = 0x6A09E667F3BCD;
            when 33 result = 0x6DFB23C651A2F;
            when 34 result = 0x71F75E8EC5F74;
            when 35 result = 0x75FEB564267C9;
            when 36 result = 0x7A11473EB0187;
            when 37 result = 0x7E2F336CF4E62;
            when 38 result = 0x82589994CCE13;
            when 39 result = 0x868D99B4492ED;
            when 40 result = 0x8ACE5422AA0DB;
            when 41 result = 0x8F1AE99157736;
            when 42 result = 0x93737B0CDC5E5;
            when 43 result = 0x97D829FDE4E50;
            when 44 result = 0x9C49182A3F090;
            when 45 result = 0xA0C667B5DE565;
            when 46 result = 0xA5503B23E255D;
            when 47 result = 0xA9E6B5579FDBF;
            when 48 result = 0xAE89F995AD3AD;
            when 49 result = 0xB33A2B84F15FB;
            when 50 result = 0xB7F76F2FB5E47;
            when 51 result = 0xBCC1E904BC1D2;
            when 52 result = 0xC199BDD85529C;
            when 53 result = 0xC67F12E57D14B;
            when 54 result = 0xCB720DCEF9069;
            when 55 result = 0xD072D4A07897C;
            when 56 result = 0xD5818DCFBA487;
            when 57 result = 0xDA9E603DB3285;
            when 58 result = 0xDFC97337B9B5F;
            when 59 result = 0xE502EE78B3FF6;
            when 60 result = 0xEA4AFA2A490DA;
            when 61 result = 0xEFA1BEE615A27;
            when 62 result = 0xF50765B6E4540;
            when 63 result = 0xFA7C1819E90D8;

    return result<N-1:0>;
```

Library pseudocode for aarch64/functions/sve/FPLogB

```
// FPLogB()
// ========

bits(N) FPLogB(bits(N) op, FPCRType fpcr)
    assert N IN {16,32,64};
    integer result;
    (fptype,sign,value) = FPUnpack(op, fpcr);

    if fptype == FPType_SNaN || fptype == FPType_QNaN || fptype == FPType_Zero then
        FPProcessException(FPExc_InvalidOp, fpcr);
        result = -(2^(N-1));            // MinInt, 100..00
    elsif fptype == FPType_Infinity then
        result = 2^(N-1) - 1;           // MaxInt, 011..11
    else
        // FPUnpack has already scaled a subnormal input
        value = Abs(value);
        result = 0;
        while value < 1.0 do
            value = value * 2.0;
            result = result - 1;
        while value >= 2.0 do
            value = value / 2.0;
            result = result + 1;

        FPProcessDenorm(fptype, N, fpcr);

    return result<N-1:0>;
```

Library pseudocode for aarch64/functions/sve/FPMinNormal

```
// FPMinNormal()
// =============

bits(N) FPMinNormal(bit sign, integer N)
    assert N IN {16,32,64};
    constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
    constant integer F = N - (E + 1);
    exp = Zeros(E-1):'1';
    frac = Zeros(F);
    return sign : exp : frac;
```

Library pseudocode for aarch64/functions/sve/FPOne

```
// FPOne()
// =======

bits(N) FPOne(bit sign, integer N)
    assert N IN {16,32,64};
    constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
    constant integer F = N - (E + 1);
    exp = '0':Ones(E-1);
    frac = Zeros(F);
    return sign : exp : frac;
```

Library pseudocode for aarch64/functions/sve/FPPointFive

```
// FPPointFive()
// =============

bits(N) FPPointFive(bit sign, integer N)
    assert N IN {16,32,64};
    constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
    constant integer F = N - (E + 1);
    exp = '0':Ones(E-2):'0';
    frac = Zeros(F);
    return sign : exp : frac;
```

Library pseudocode for aarch64/functions/sve/FPScale

```
// FPScale()
// =========

bits(N) FPScale(bits (N) op, integer scale, FPCRType fpcr)
    assert N IN {16,32,64};
    bits(N) result;
    (fptype,sign,value) = FPUnpack(op, fpcr);

    if fptype == FPType_SNaN || fptype == FPType_QNaN then
        result = FPProcessNaN(fptype, op, fpcr);
    elsif fptype == FPType_Zero then
        result = FPZero(sign, N);
    elsif fptype == FPType_Infinity then
        result = FPInfinity(sign, N);
    else
        result = FPRound(value * (2.0^scale), fpcr, N);
        FPProcessDenorm(fptype, N, fpcr);

    return result;
```

Library pseudocode for aarch64/functions/sve/FPTrigMAdd

```
// FPTrigMAdd()
// ============

bits(N) FPTrigMAdd(integer x_in, bits(N) op1, bits(N) op2_in, FPCRType fpcr)
    assert N IN {16,32,64};
    bits(N) coeff;
    bits(N) op2 = op2_in;
    integer x = x_in;
    assert x >= 0;
    assert x < 8;

    if op2<N-1> == '1' then
        x = x + 8;

    coeff    = FPTrigMAddCoefficient[x, N];
    op2      = FPAbs(op2);
    result   = FPMulAdd(coeff, op1, op2, fpcr);
    return result;
```

Library pseudocode for aarch64/functions/sve/FPTrigMAddCoefficient

```
// FPTrigMAddCoefficient()
// =======================

bits(N) FPTrigMAddCoefficient[integer index, integer N]
    assert N IN {16,32,64};
    integer result;

    if N == 16 then
        case index of
            when  0 result = 0x3c00;
            when  1 result = 0xb155;
            when  2 result = 0x2030;
            when  3 result = 0x0000;
            when  4 result = 0x0000;
            when  5 result = 0x0000;
            when  6 result = 0x0000;
            when  7 result = 0x0000;
            when  8 result = 0x3c00;
            when  9 result = 0xb800;
            when 10 result = 0x293a;
            when 11 result = 0x0000;
            when 12 result = 0x0000;
            when 13 result = 0x0000;
            when 14 result = 0x0000;
            when 15 result = 0x0000;
    elsif N == 32 then
        case index of
            when  0 result = 0x3f800000;
            when  1 result = 0xbe2aaaab;
            when  2 result = 0x3c088886;
            when  3 result = 0xb95008b9;
            when  4 result = 0x36369d6d;
            when  5 result = 0x00000000;
            when  6 result = 0x00000000;
            when  7 result = 0x00000000;
            when  8 result = 0x3f800000;
            when  9 result = 0xbf000000;
            when 10 result = 0x3d2aaaa6;
            when 11 result = 0xbab60705;
            when 12 result = 0x37cd37cc;
            when 13 result = 0x00000000;
            when 14 result = 0x00000000;
            when 15 result = 0x00000000;
    else // N == 64
        case index of
            when  0 result = 0x3ff0000000000000;
            when  1 result = 0xbfc5555555555543;
            when  2 result = 0x3f8111111110f30c;
            when  3 result = 0xbf2a01a019b92fc6;
            when  4 result = 0x3ec71de351f3d22b;
            when  5 result = 0xbe5ae5e2b60f7b91;
            when  6 result = 0x3de5d8408868552f;
            when  7 result = 0x0000000000000000;
            when  8 result = 0x3ff0000000000000;
            when  9 result = 0xbfe0000000000000;
            when 10 result = 0x3fa5555555555536;
            when 11 result = 0xbf56c16c16c13a0b;
            when 12 result = 0x3efa01a019b1e8d8;
            when 13 result = 0xbe927e4f7282f468;
            when 14 result = 0x3e21ee96d2641b13;
            when 15 result = 0xbda8f76380fbb401;

    return result<N-1:0>;
```

Library pseudocode for aarch64/functions/sve/FPTrigSMul

```
// FPTrigSMul()
// ============

bits(N) FPTrigSMul(bits(N) op1, bits(N) op2, FPCRType fpcr)
    assert N IN {16,32,64};
    result = FPMul(op1, op1, fpcr);
    fpexc = FALSE;
    (fptype, sign, value) = FPUnpack(result, fpcr, fpexc);

    if !(fptype IN {FPType_QNaN, FPType_SNaN}) then
        result<N-1> = op2<0>;

    return result;
```

Library pseudocode for aarch64/functions/sve/FPTrigSSel

```
// FPTrigSSel()
// ============

bits(N) FPTrigSSel(bits(N) op1, bits(N) op2)
    assert N IN {16,32,64};
    bits(N) result;

    if op2<0> == '1' then
        result = FPOne(op2<1>, N);
    elsif op2<1> == '1' then
        result = FPNeg(op1);
    else
        result = op1;

    return result;
```

Library pseudocode for aarch64/functions/sve/FirstActive

```
// FirstActive()
// =============

bit FirstActive(bits(N) mask, bits(N) x, integer esize)
    integer elements = N DIV (esize DIV 8);
    for e = 0 to elements-1
        if ActivePredicateElement(mask, e, esize) then
            return PredicateElement(x, e, esize);
    return '0';
```

Library pseudocode for aarch64/functions/sve/FloorPow2

```
// FloorPow2()
// ===========
// For a positive integer X, return the largest power of 2 <= X

integer FloorPow2(integer x)
    assert x >= 0;
    integer n = 1;
    if x == 0 then return 0;
    while x >= 2^n do
        n = n + 1;
    return 2^(n - 1);
```

Library pseudocode for aarch64/functions/sve/HaveSMEFullA64

```
// HaveSMEFullA64()
// ================
// Returns TRUE if the SME FA64 extension is implemented, FALSE otherwise.

boolean HaveSMEFullA64()
    return IsFeatureImplemented(FEAT_SME_FA64);
```

Library pseudocode for aarch64/functions/sve/HaveSVE

```
// HaveSVE()
// =========

boolean HaveSVE()
    return IsFeatureImplemented(FEAT_SVE);
```

Library pseudocode for aarch64/functions/sve/HaveSVE2

```
// HaveSVE2()
// ==========
// Returns TRUE if the SVE2 extension is implemented, FALSE otherwise.

boolean HaveSVE2()
    return IsFeatureImplemented(FEAT_SVE2);
```

Library pseudocode for aarch64/functions/sve/HaveSVE2AES

```
// HaveSVE2AES()
// =============
// Returns TRUE if the SVE2 AES extension is implemented, FALSE otherwise.

boolean HaveSVE2AES()
    return IsFeatureImplemented(FEAT_SVE_AES);
```

Library pseudocode for aarch64/functions/sve/HaveSVE2BitPerm

```
// HaveSVE2BitPerm()
// =================
// Returns TRUE if the SVE2 Bit Permissions extension is implemented, FALSE otherwise.

boolean HaveSVE2BitPerm()
    return IsFeatureImplemented(FEAT_SVE_BitPerm);
```

Library pseudocode for aarch64/functions/sve/HaveSVE2PMULL128

```
// HaveSVE2PMULL128()
// ==================
// Returns TRUE if the SVE2 128 bit PMULL extension is implemented, FALSE otherwise.

boolean HaveSVE2PMULL128()
    return IsFeatureImplemented(FEAT_SVE_PMULL128);
```

Library pseudocode for aarch64/functions/sve/HaveSVE2SHA256

```
// HaveSVE2SHA256()
// ================
// Returns TRUE if the SVE2 SHA256 extension is implemented, FALSE otherwise.

boolean HaveSVE2SHA256()
    return HaveSVE2() && boolean IMPLEMENTATION_DEFINED "Have SVE2 SHA256 extension";
```

Library pseudocode for aarch64/functions/sve/HaveSVE2SHA3

```
// HaveSVE2SHA3()
// ==============
// Returns TRUE if the SVE2 SHA3 extension is implemented, FALSE otherwise.

boolean HaveSVE2SHA3()
    return IsFeatureImplemented(FEAT_SVE_SHA3);
```

Library pseudocode for aarch64/functions/sve/HaveSVE2SHA512

```
// HaveSVE2SHA512()
// ================
// Returns TRUE if the SVE2 SHA512 extension is implemented, FALSE otherwise.

boolean HaveSVE2SHA512()
    return HaveSVE2() && boolean IMPLEMENTATION_DEFINED "Have SVE2 SHA512 extension";
```

Library pseudocode for aarch64/functions/sve/HaveSVE2SM3

```
// HaveSVE2SM3()
// =============
// Returns TRUE if the SVE2 SM3 extension is implemented, FALSE otherwise.

boolean HaveSVE2SM3()
    return HaveSVE2() && boolean IMPLEMENTATION_DEFINED "Have SVE2 SM3 extension";
```

Library pseudocode for aarch64/functions/sve/HaveSVE2SM4

```
// HaveSVE2SM4()
// =============
// Returns TRUE if the SVE2 SM4 extension is implemented, FALSE otherwise.

boolean HaveSVE2SM4()
    return IsFeatureImplemented(FEAT_SVE_SM4);
```

Library pseudocode for aarch64/functions/sve/HaveSVE2p1

```
// HaveSVE2p1()
// ============
// Returns TRUE if the SVE2.1 extension is implemented, FALSE otherwise.

boolean HaveSVE2p1()
    return IsFeatureImplemented(FEAT_SVE2p1);
```

Library pseudocode for aarch64/functions/sve/HaveSVEB16B16

```
// HaveSVEB16B16()
// ===============
// Returns TRUE if the SVE2.1 non-widening BFloat16 instructions are implemented, FALSE otherwise.

boolean HaveSVEB16B16()
    return IsFeatureImplemented(FEAT_SVE_B16B16);
```

Library pseudocode for aarch64/functions/sve/HaveSVEFP32MatMulExt

```
// HaveSVEFP32MatMulExt()
// ======================
// Returns TRUE if single-precision floating-point matrix multiply instruction support implemented
// and FALSE otherwise.

boolean HaveSVEFP32MatMulExt()
    return IsFeatureImplemented(FEAT_F32MM);
```

Library pseudocode for aarch64/functions/sve/HaveSVEFP64MatMulExt

```
// HaveSVEFP64MatMulExt()
// ======================
// Returns TRUE if double-precision floating-point matrix multiply instruction support implemented
// and FALSE otherwise.

boolean HaveSVEFP64MatMulExt()
    return IsFeatureImplemented(FEAT_F64MM);
```

Library pseudocode for aarch64/functions/sve/ImplementedSMEVectorLength

```
// ImplementedSMEVectorLength()
// ============================
// Reduce SVE/SME vector length to a supported value (power of two)

integer ImplementedSMEVectorLength(integer nbits_in)
    integer maxbits = MaxImplementedSVL();
    assert 128 <= maxbits && maxbits <= 2048 && IsPow2(maxbits);
    integer nbits = Min(nbits_in, maxbits);
    assert 128 <= nbits && nbits <= 2048 && Align(nbits, 128) == nbits;

    // Search for a supported power-of-two VL less than or equal to nbits
    while nbits > 128 do
        if IsPow2(nbits) && SupportedPowerTwoSVL(nbits) then return nbits;
        nbits = nbits - 128;

    // Return the smallest supported power-of-two VL
    nbits = 128;
    while nbits < maxbits do
        if SupportedPowerTwoSVL(nbits) then return nbits;
        nbits = nbits * 2;

    // The only option is maxbits
    return maxbits;
```

Library pseudocode for aarch64/functions/sve/ImplementedSVEVectorLength

```
// ImplementedSVEVectorLength()
// ============================
// Reduce SVE vector length to a supported value (power of two)

integer ImplementedSVEVectorLength(integer nbits_in)
    integer maxbits = MaxImplementedVL();
    assert 128 <= maxbits && maxbits <= 2048 && IsPow2(maxbits);
    integer nbits = Min(nbits_in, maxbits);
    assert 128 <= nbits && nbits <= 2048 && Align(nbits, 128) == nbits;

    while nbits > 128 do
        if IsPow2(nbits) then return nbits;
        nbits = nbits - 128;
    return nbits;
```

Library pseudocode for aarch64/functions/sve/InStreamingMode

```
// InStreamingMode()
// =================

boolean InStreamingMode()
    return HaveSME() && PSTATE.SM == '1';
```

Library pseudocode for aarch64/functions/sve/IsEven

```
// IsEven()
// ========

boolean IsEven(integer val)
    return val MOD 2 == 0;
```

Library pseudocode for aarch64/functions/sve/IsFPEnabled

```
// IsFPEnabled()
// =============
// Returns TRUE if accesses to the Advanced SIMD and floating-point
// registers are enabled at the target exception level in the current
// execution state and FALSE otherwise.

boolean IsFPEnabled(bits(2) el)
    if ELUsingAArch32(el) then
        return AArch32.IsFPEnabled(el);
    else
        return AArch64.IsFPEnabled(el);
```

Library pseudocode for aarch64/functions/sve/IsFullA64Enabled

```
// IsFullA64Enabled()
// ==================
// Returns TRUE is full A64 is enabled in Streaming mode and FALSE othersise.

boolean IsFullA64Enabled()
    if !HaveSMEFullA64() then return FALSE;

    // Check if full SVE disabled in SMCR_EL1
    if PSTATE.EL IN {EL0, EL1} && !IsInHost() then
        // Check full SVE at EL0/EL1
        if SMCR_EL1.FA64 == '0' then return FALSE;

    // Check if full SVE disabled in SMCR_EL2
    if PSTATE.EL IN {EL0, EL1, EL2} && EL2Enabled() then
        if SMCR_EL2.FA64 == '0' then return FALSE;

    // Check if full SVE disabled in SMCR_EL3
    if HaveEL(EL3) then
        if SMCR_EL3.FA64 == '0' then return FALSE;

    return TRUE;
```

Library pseudocode for aarch64/functions/sve/IsOdd

```
// IsOdd()
// =======

boolean IsOdd(integer val)
    return val MOD 2 == 1;
```

Library pseudocode for aarch64/functions/sve/IsOriginalSVEEnabled

```
// IsOriginalSVEEnabled()
// ======================
// Returns TRUE if access to SVE functionality is enabled at the target
// exception level and FALSE otherwise.

boolean IsOriginalSVEEnabled(bits(2) el)
    boolean disabled;
    if ELUsingAArch32(el) then
        return FALSE;

    // Check if access disabled in CPACR_EL1
    if el IN {EL0, EL1} && !IsInHost() then
        // Check SVE at EL0/EL1
        case CPACR_EL1.ZEN of
            when 'x0' disabled = TRUE;
            when '01' disabled = el == EL0;
            when '11' disabled = FALSE;
        if disabled then return FALSE;

    // Check if access disabled in CPTR_EL2
    if el IN {EL0, EL1, EL2} && EL2Enabled() then
        if HaveVirtHostExt() && HCR_EL2.E2H == '1' then
            case CPTR_EL2.ZEN of
                when 'x0' disabled = TRUE;
                when '01' disabled = el == EL0 && HCR_EL2.TGE == '1';
                when '11' disabled = FALSE;
            if disabled then return FALSE;
        else
            if CPTR_EL2.TZ == '1' then return FALSE;

    // Check if access disabled in CPTR_EL3
    if HaveEL(EL3) then
        if CPTR_EL3.EZ == '0' then return FALSE;

    return TRUE;
```

Library pseudocode for aarch64/functions/sve/IsPow2

```
// IsPow2()
// ========
// Return TRUE if positive integer X is a power of 2. Otherwise,
// return FALSE.

boolean IsPow2(integer x)
    if x <= 0 then return FALSE;
    return FloorPow2(x) == CeilPow2(x);
```

Library pseudocode for aarch64/functions/sve/IsSMEEnabled

```
// IsSMEEnabled()
// ==============
// Returns TRUE if access to SME functionality is enabled at the target
// exception level and FALSE otherwise.

boolean IsSMEEnabled(bits(2) el)
    boolean disabled;
    if ELUsingAArch32(el) then
        return FALSE;

    // Check if access disabled in CPACR_EL1
    if el IN {EL0, EL1} && !IsInHost() then
        // Check SME at EL0/EL1
        case CPACR_EL1.SMEN of
            when 'x0' disabled = TRUE;
            when '01' disabled = el == EL0;
            when '11' disabled = FALSE;
        if disabled then return FALSE;

    // Check if access disabled in CPTR_EL2
    if el IN {EL0, EL1, EL2} && EL2Enabled() then
        if HaveVirtHostExt() && HCR_EL2.E2H == '1' then
            case CPTR_EL2.SMEN of
                when 'x0' disabled = TRUE;
                when '01' disabled = el == EL0 && HCR_EL2.TGE == '1';
                when '11' disabled = FALSE;
            if disabled then return FALSE;
        else
            if CPTR_EL2.TSM == '1' then return FALSE;

    // Check if access disabled in CPTR_EL3
    if HaveEL(EL3) then
        if CPTR_EL3.ESM == '0' then return FALSE;

    return TRUE;
```

Library pseudocode for aarch64/functions/sve/IsSVEEnabled

```
// IsSVEEnabled()
// ==============
// Returns TRUE if access to SVE registers is enabled at the target exception
// level and FALSE otherwise.

boolean IsSVEEnabled(bits(2) el)
    if HaveSME() && PSTATE.SM == '1' then
        return IsSMEEnabled(el);
    elsif HaveSVE() then
        return IsOriginalSVEEnabled(el);
    else
        return FALSE;
```

Library pseudocode for aarch64/functions/sve/LastActive

```
// LastActive()
// ============

bit LastActive(bits(N) mask, bits(N) x, integer esize)
    integer elements = N DIV (esize DIV 8);
    for e = elements-1 downto 0
        if ActivePredicateElement(mask, e, esize) then
            return PredicateElement(x, e, esize);
    return '0';
```

Library pseudocode for aarch64/functions/sve/LastActiveElement

```
// LastActiveElement()
// ===================

integer LastActiveElement(bits(N) mask, integer esize)
    integer elements = N DIV (esize DIV 8);
    for e = elements-1 downto 0
        if ActivePredicateElement(mask, e, esize) then return e;
    return -1;
```

Library pseudocode for aarch64/functions/sve/MaxImplementedSVL

```
// MaxImplementedSVL()
// ===================

integer MaxImplementedSVL()
    return integer IMPLEMENTATION_DEFINED "Max implemented SVL";
```

Library pseudocode for aarch64/functions/sve/MaxImplementedVL

```
// MaxImplementedVL()
// ==================

integer MaxImplementedVL()
    return integer IMPLEMENTATION_DEFINED "Max implemented VL";
```

Library pseudocode for aarch64/functions/sve/MaybeZeroSVEUppers

```
// MaybeZeroSVEUppers()
// ====================

MaybeZeroSVEUppers(bits(2) target_el)
    boolean lower_enabled;

    if UInt(target_el) <= UInt(PSTATE.EL) || !IsSVEEnabled(target_el) then
        return;

    if target_el == EL3 then
        if EL2Enabled() then
            lower_enabled = IsFPEnabled(EL2);
        else
            lower_enabled = IsFPEnabled(EL1);
    elsif target_el == EL2 then
        assert !ELUsingAArch32(EL2);
        if HCR_EL2.TGE == '0' then
            lower_enabled = IsFPEnabled(EL1);
        else
            lower_enabled = IsFPEnabled(EL0);
    else
        assert target_el == EL1 && !ELUsingAArch32(EL1);
        lower_enabled = IsFPEnabled(EL0);

    if lower_enabled then
        constant integer VL = if IsSVEEnabled(PSTATE.EL) then CurrentVL else 128;
        constant integer PL = VL DIV 8;
        for n = 0 to 31
            if ConstrainUnpredictableBool(Unpredictable_SVEZEROUPPER) then
                _Z[n] = ZeroExtend(_Z[n]<VL-1:0>, MAX_VL);
        for n = 0 to 15
            if ConstrainUnpredictableBool(Unpredictable_SVEZEROUPPER) then
                _P[n] = ZeroExtend(_P[n]<PL-1:0>, MAX_PL);
        if ConstrainUnpredictableBool(Unpredictable_SVEZEROUPPER) then
            _FFR = ZeroExtend(_FFR<PL-1:0>, MAX_PL);
        if HaveSME() && PSTATE.ZA == '1' then
            constant integer SVL = CurrentSVL;
            constant integer accessiblevecs = SVL DIV 8;
            constant integer allvecs = MaxImplementedSVL() DIV 8;

            for n = 0 to accessiblevecs - 1
                if ConstrainUnpredictableBool(Unpredictable_SMEZEROUPPER) then
                    _ZA[n] = ZeroExtend(_ZA[n]<SVL-1:0>, MAX_VL);
            for n = accessiblevecs to allvecs - 1
                if ConstrainUnpredictableBool(Unpredictable_SMEZEROUPPER) then
                    _ZA[n] = Zeros(MAX_VL);
```

Library pseudocode for aarch64/functions/sve/MemNF

```
// MemNF[] - non-assignment form
// =============================

(bits(8*size), boolean) MemNF[bits(64) address, integer size, AccessDescriptor accdesc]
    assert size IN {1, 2, 4, 8, 16};
    bits(8*size) value;
    boolean bad;

    boolean aligned = IsAligned(address, size);

    if !aligned && AlignmentEnforced() then
        return (bits(8*size) UNKNOWN, TRUE);

    boolean atomic = aligned || size == 1;

    if !atomic then
        (value<7:0>, bad) = MemSingleNF[address, 1, accdesc, aligned];

        if bad then
            return (bits(8*size) UNKNOWN, TRUE);

        // For subsequent bytes it is CONSTRAINED UNPREDICTABLE whether an unaligned Device memory
        // access will generate an Alignment Fault, as to get this far means the first byte did
        // not, so we must be changing to a new translation page.
        if !aligned then
            c = ConstrainUnpredictable(Unpredictable_DEVPAGE2);
            assert c IN {Constraint_FAULT, Constraint_NONE};
            if c == Constraint_NONE then aligned = TRUE;

        for i = 1 to size-1
            (value<8*i+7:8*i>, bad) = MemSingleNF[address+i, 1, accdesc, aligned];

            if bad then
                return (bits(8*size) UNKNOWN, TRUE);
    else
        (value, bad) = MemSingleNF[address, size, accdesc, aligned];
        if bad then
            return (bits(8*size) UNKNOWN, TRUE);

    if BigEndian(accdesc.acctype) then
        value = BigEndianReverse(value);

    return (value, FALSE);
```

Library pseudocode for aarch64/functions/sve/MemSingleNF

```
// MemSingleNF[] - non-assignment form
// ===================================

(bits(8*size), boolean) MemSingleNF[bits(64) address, integer size, AccessDescriptor accdesc_in,
                                    boolean aligned]
    assert accdesc_in.acctype == AccessType_SVE;
    assert accdesc_in.nonfault || (accdesc_in.firstfault && !accdesc_in.first);

    bits(8*size) value;
    AddressDescriptor memaddrdesc;
    PhysMemRetStatus memstatus;
    AccessDescriptor accdesc = accdesc_in;
    FaultRecord fault = NoFault(accdesc);

    // Implementation may suppress NF load for any reason
    if ConstrainUnpredictableBool(Unpredictable_NONFAULT) then
        return (bits(8*size) UNKNOWN, TRUE);

    // If the instruction encoding permits tag checking, confer with system register configuration
    // which may override this.
    if HaveMTE2Ext() && accdesc.tagchecked then
        accdesc.tagchecked = AArch64.AccessIsTagChecked(address, accdesc);

    // MMU or MPU
    memaddrdesc = AArch64.TranslateAddress(address, accdesc, aligned, size);

    // Non-fault load from Device memory must not be performed externally
    if memaddrdesc.memattrs.memtype == MemType_Device then
        return (bits(8*size) UNKNOWN, TRUE);

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        return (bits(8*size) UNKNOWN, TRUE);

    if HaveMTE2Ext() && accdesc.tagchecked then
        bits(4) ptag = AArch64.PhysicalTag(address);
        if !AArch64.CheckTag(memaddrdesc, accdesc, ptag) then
            return (bits(8*size) UNKNOWN, TRUE);

    (memstatus, value) = PhysMemRead(memaddrdesc, size, accdesc);
    if IsFault(memstatus) then
        boolean iswrite = FALSE;
        if IsExternalAbortTakenSynchronously(memstatus, iswrite, memaddrdesc, size, accdesc) then
            return (bits(8*size) UNKNOWN, TRUE);
        fault.merrorstate = memstatus.merrorstate;
        fault.extflag    = memstatus.extflag;
        fault.statuscode = memstatus.statuscode;
        PendSErrorInterrupt(fault);

    return (value, FALSE);
```

Library pseudocode for aarch64/functions/sve/NoneActive

```
// NoneActive()
// ============

bit NoneActive(bits(N) mask, bits(N) x, integer esize)
    integer elements = N DIV (esize DIV 8);
    for e = 0 to elements-1
        if ActivePredicateElement(mask, e, esize) && ActivePredicateElement(x, e, esize) then
            return '0';
    return '1';
```

Library pseudocode for aarch64/functions/sve/P

```
// P[] - non-assignment form
// =========================

bits(width) P[integer n, integer width]
    assert n >= 0 && n <= 31;
    assert width == CurrentVL DIV 8;
    return _P[n]<width-1:0>;

// P[] - assignment form
// =====================

P[integer n, integer width] = bits(width) value
    assert n >= 0 && n <= 31;
    assert width == CurrentVL DIV 8;
    if ConstrainUnpredictableBool(Unpredictable_SVEZEROUPPER) then
        _P[n] = ZeroExtend(value, MAX_PL);
    else
        _P[n]<width-1:0> = value;
```

Library pseudocode for aarch64/functions/sve/PredTest

```
// PredTest()
// ==========

bits(4) PredTest(bits(N) mask, bits(N) result, integer esize)
    bit n = FirstActive(mask, result, esize);
    bit z = NoneActive(mask, result, esize);
    bit c = NOT LastActive(mask, result, esize);
    bit v = '0';
    return n:z:c:v;
```

Library pseudocode for aarch64/functions/sve/PredicateElement

```
// PredicateElement()
// ==================
// Returns the predicate bit

bit PredicateElement(bits(N) pred, integer e, integer esize)
    assert esize IN {8, 16, 32, 64, 128};
    integer n = e * (esize DIV 8);
    assert n >= 0 && n < N;
    return pred<n>;
```

Library pseudocode for aarch64/functions/sve/ReducePredicated

```
// ReducePredicated()
// ==================

bits(esize) ReducePredicated(ReduceOp op, bits(N) input, bits(M) mask, bits(esize) identity)
    assert(N == M * 8);
    integer p2bits = CeilPow2(N);
    bits(p2bits) operand;
    integer elements = p2bits DIV esize;

    for e = 0 to elements-1
        if e * esize < N && ActivePredicateElement(mask, e, esize) then
            Elem[operand, e, esize] = Elem[input, e, esize];
        else
            Elem[operand, e, esize] = identity;

    return Reduce(op, operand, esize);
```

Library pseudocode for aarch64/functions/sve/ResetSMEState

```
// ResetSMEState()
// ===============

ResetSMEState()
    integer vectors = MAX_VL DIV 8;
    for n = 0 to vectors - 1
        _ZA[n] = Zeros(MAX_VL);
    _ZT0 = Zeros(ZT0_LEN);
```

Library pseudocode for aarch64/functions/sve/ResetSVEState

```
// ResetSVEState()
// ===============

ResetSVEState()
    for n = 0 to 31
        _Z[n] = Zeros(MAX_VL);
    for n = 0 to 15
        _P[n] = Zeros(MAX_PL);
    _FFR = Zeros(MAX_PL);
    FPSR = ZeroExtend(0x0800009f<31:0>, 64);
```

Library pseudocode for aarch64/functions/sve/Reverse

```
// Reverse()
// =========
// Reverse subwords of M bits in an N-bit word

bits(N) Reverse(bits(N) word, integer M)
    bits(N) result;
    integer sw = N DIV M;
    assert N == sw * M;
    for s = 0 to sw-1
        Elem[result, (sw - 1) - s, M] = Elem[word, s, M];
    return result;
```

Library pseudocode for aarch64/functions/sve/SMEAccessTrap

```
// SMEAccessTrap()
// ===============
// Trapped access to SME registers due to CPACR_EL1, CPTR_EL2, or CPTR_EL3.

SMEAccessTrap(SMEExceptionType etype, bits(2) target_el_in)
    bits(2) target_el = target_el_in;
    assert UInt(target_el) >= UInt(PSTATE.EL);
    if target_el == EL0 then
        target_el = EL1;
    boolean route_to_el2;
    route_to_el2 = PSTATE.EL == EL0 && target_el == EL1 && EL2Enabled() && HCR_EL2.TGE == '1';

    except = ExceptionSyndrome(Exception_SMEAccessTrap);
    bits(64) preferred_exception_return = ThisInstrAddr(64);
    vect_offset = 0x0;

    case etype of
        when SMEExceptionType_AccessTrap
            except.syndrome<2:0> = '000';
        when SMEExceptionType_Streaming
            except.syndrome<2:0> = '001';
        when SMEExceptionType_NotStreaming
            except.syndrome<2:0> = '010';
        when SMEExceptionType_InactiveZA
            except.syndrome<2:0> = '011';
        when SMEExceptionType_InaccessibleZT0
            except.syndrome<2:0> = '100';

    if route_to_el2 then
        AArch64.TakeException(EL2, except, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(target_el, except, preferred_exception_return, vect_offset);
```

Library pseudocode for aarch64/functions/sve/SMEExceptionType

```
// SMEExceptionType
// ================
enumeration SMEExceptionType {
    SMEExceptionType_AccessTrap,        // SME functionality trapped or disabled
    SMEExceptionType_Streaming,         // Illegal instruction in Streaming SVE mode
    SMEExceptionType_NotStreaming,      // Illegal instruction not in Streaming SVE mode
    SMEExceptionType_InactiveZA,        // Illegal instruction when ZA is inactive
    SMEExceptionType_InaccessibleZT0,   // Access to ZT0 is disabled
};
```

Library pseudocode for aarch64/functions/sve/SVEAccessTrap

```
// SVEAccessTrap()
// ===============
// Trapped access to SVE registers due to CPACR_EL1, CPTR_EL2, or CPTR_EL3.

SVEAccessTrap(bits(2) target_el)
    assert UInt(target_el) >= UInt(PSTATE.EL) && target_el != EL0 && HaveEL(target_el);
    route_to_el2 = target_el == EL1 && EL2Enabled() && HCR_EL2.TGE == '1';

    except = ExceptionSyndrome(Exception_SVEAccessTrap);
    bits(64) preferred_exception_return = ThisInstrAddr(64);
    vect_offset = 0x0;

    if route_to_el2 then
        AArch64.TakeException(EL2, except, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(target_el, except, preferred_exception_return, vect_offset);
```

Library pseudocode for aarch64/functions/sve/SVECmp

```
// SVECmp
// ======

enumeration SVECmp { Cmp_EQ, Cmp_NE, Cmp_GE, Cmp_GT, Cmp_LT, Cmp_LE, Cmp_UN };
```

Library pseudocode for aarch64/functions/sve/SVEMoveMaskPreferred

```
// SVEMoveMaskPreferred()
// ======================
// Return FALSE if a bitmask immediate encoding would generate an immediate
// value that could also be represented by a single DUP instruction.
// Used as a condition for the preferred MOV<-DUPM alias.

boolean SVEMoveMaskPreferred(bits(13) imm13)
    bits(64) imm;
    (imm, -) = DecodeBitMasks(imm13<12>, imm13<5:0>, imm13<11:6>, TRUE, 64);

    // Check for 8 bit immediates
    if !IsZero(imm<7:0>) then
        // Check for 'ffffffffffffffxy' or '00000000000000xy'
        if IsZero(imm<63:7>) || IsOnes(imm<63:7>) then
            return FALSE;

        // Check for 'ffffffxyffffffxy' or '000000xy000000xy'
        if imm<63:32> == imm<31:0> && (IsZero(imm<31:7>) || IsOnes(imm<31:7>)) then
            return FALSE;

        // Check for 'ffxyffxyffxyffxy' or '00xy00xy00xy00xy'
        if (imm<63:32> == imm<31:0> && imm<31:16> == imm<15:0> &&
              (IsZero(imm<15:7>) || IsOnes(imm<15:7>))) then
            return FALSE;

        // Check for 'xyxyxyxyxyxyxyxy'
        if imm<63:32> == imm<31:0> && imm<31:16> == imm<15:0> && (imm<15:8> == imm<7:0>) then
            return FALSE;

    // Check for 16 bit immediates
    else
        // Check for 'ffffffffffffxy00' or '000000000000xy00'
        if IsZero(imm<63:15>) || IsOnes(imm<63:15>) then
            return FALSE;

        // Check for 'ffffxy00ffffxy00' or '0000xy000000xy00'
        if imm<63:32> == imm<31:0> && (IsZero(imm<31:7>) || IsOnes(imm<31:7>)) then
            return FALSE;

        // Check for 'xy00xy00xy00xy00'
        if imm<63:32> == imm<31:0> && imm<31:16> == imm<15:0> then
            return FALSE;

    return TRUE;
```

Library pseudocode for aarch64/functions/sve/SetPSTATE_SM

```
// SetPSTATE_SM()
// ==============

SetPSTATE_SM(bit value)
    if PSTATE.SM != value then
        ResetSVEState();
        PSTATE.SM = value;
```

Library pseudocode for aarch64/functions/sve/SetPSTATE_SVCR

```
// SetPSTATE_SVCR
// ==============

SetPSTATE_SVCR(bits(32) svcr)
    SetPSTATE_SM(svcr<0>);
    SetPSTATE_ZA(svcr<1>);
```

Library pseudocode for aarch64/functions/sve/SetPSTATE_ZA

```
// SetPSTATE_ZA()
// ==============

SetPSTATE_ZA(bit value)
    if PSTATE.ZA != value then
        ResetSMEState();
        PSTATE.ZA = value;
```

Library pseudocode for aarch64/functions/sve/ShiftSat

```
// ShiftSat()
// ==========

integer ShiftSat(integer shift, integer esize)
    if shift > esize+1 then return esize+1;
    elsif shift < -(esize+1) then return -(esize+1);
    return shift;
```

Library pseudocode for aarch64/functions/sve/SupportedPowerTwoSVL

```
// SupportedPowerTwoSVL()
// ======================
// Return an IMPLEMENTATION DEFINED specific value
// returns TRUE if SVL is supported and is a power of two, FALSE otherwise

boolean SupportedPowerTwoSVL(integer nbits);
```

Library pseudocode for aarch64/functions/sve/System

```
constant integer MAX_VL = 2048;
constant integer MAX_PL = 256;
constant integer ZT0_LEN =  512;
bits(MAX_PL) _FFR;

array bits(MAX_VL) _Z[0..31];

array bits(MAX_PL) _P[0..15];
```

Library pseudocode for aarch64/functions/sve/Z

```
// Z[] - non-assignment form
// =========================

bits(width) Z[integer n, integer width]
    assert n >= 0 && n <= 31;
    assert width == CurrentVL;
    return _Z[n]<width-1:0>;

// Z[] - assignment form
// =====================

Z[integer n, integer width] = bits(width) value
    assert n >= 0 && n <= 31;
    assert width == CurrentVL;
    if ConstrainUnpredictableBool(Unpredictable_SVEZEROUPPER) then
        _Z[n] = ZeroExtend(value, MAX_VL);
    else
        _Z[n]<width-1:0> = value;
```

Library pseudocode for aarch64/functions/syshintop/SystemHintOp

```
// SystemHintOp
// ============
// System Hint instruction types.

enumeration SystemHintOp {
    SystemHintOp_NOP,
    SystemHintOp_YIELD,
    SystemHintOp_WFE,
    SystemHintOp_WFI,
    SystemHintOp_SEV,
    SystemHintOp_SEVL,
    SystemHintOp_DGH,
    SystemHintOp_ESB,
    SystemHintOp_PSB,
    SystemHintOp_TSB,
    SystemHintOp_BTI,
    SystemHintOp_WFET,
    SystemHintOp_WFIT,
    SystemHintOp_CLRBHB,
    SystemHintOp_GCSB,
    SystemHintOp_CHKFEAT,
    SystemHintOp_CSDB
};
```

Library pseudocode for aarch64/functions/sysop/SysOp

```
// SysOp()
// =======

SystemOp SysOp(bits(3) op1, bits(4) CRn, bits(4) CRm, bits(3) op2)
    case op1:CRn:CRm:op2 of
        when '000 0111 1000 000' return Sys_AT;     // S1E1R
        when '000 0111 1000 001' return Sys_AT;     // S1E1W
        when '000 0111 1000 010' return Sys_AT;     // S1E0R
        when '000 0111 1000 011' return Sys_AT;     // S1E0W
        when '000 0111 1001 000' return Sys_AT;     // S1E1RP
        when '000 0111 1001 001' return Sys_AT;     // S1E1WP
        when '100 0111 1000 000' return Sys_AT;     // S1E2R
        when '100 0111 1000 001' return Sys_AT;     // S1E2W
        when '100 0111 1000 100' return Sys_AT;     // S12E1R
        when '100 0111 1000 101' return Sys_AT;     // S12E1W
        when '100 0111 1000 110' return Sys_AT;     // S12E0R
        when '100 0111 1000 111' return Sys_AT;     // S12E0W
        when '110 0111 1000 000' return Sys_AT;     // S1E3R
        when '110 0111 1000 001' return Sys_AT;     // S1E3W
        when '001 0111 0010 100' return Sys_BRB;    // IALL
        when '001 0111 0010 101' return Sys_BRB;    // INJ
        when '000 0111 0110 001' return Sys_DC;     // IVAC
        when '000 0111 0110 010' return Sys_DC;     // ISW
        when '000 0111 0110 011' return Sys_DC;     // IGVAC
        when '000 0111 0110 100' return Sys_DC;     // IGSW
        when '000 0111 0110 101' return Sys_DC;     // IGDVAC
        when '000 0111 0110 110' return Sys_DC;     // IGDSW
        when '000 0111 1010 010' return Sys_DC;     // CSW
        when '000 0111 1010 100' return Sys_DC;     // CGSW
        when '000 0111 1010 110' return Sys_DC;     // CGDSW
        when '000 0111 1110 010' return Sys_DC;     // CISW
        when '000 0111 1110 100' return Sys_DC;     // CIGSW
        when '000 0111 1110 110' return Sys_DC;     // CIGDSW
        when '011 0111 0100 001' return Sys_DC;     // ZVA
        when '011 0111 0100 011' return Sys_DC;     // GVA
        when '011 0111 0100 100' return Sys_DC;     // GZVA
        when '011 0111 1010 001' return Sys_DC;     // CVAC
        when '011 0111 1010 011' return Sys_DC;     // CGVAC
        when '011 0111 1010 101' return Sys_DC;     // CGDVAC
        when '011 0111 1011 001' return Sys_DC;     // CVAU
        when '011 0111 1100 001' return Sys_DC;     // CVAP
        when '011 0111 1100 011' return Sys_DC;     // CGVAP
        when '011 0111 1100 101' return Sys_DC;     // CGDVAP
        when '011 0111 1101 001' return Sys_DC;     // CVADP
        when '011 0111 1101 011' return Sys_DC;     // CGVADP
        when '011 0111 1101 101' return Sys_DC;     // CGDVADP
        when '011 0111 1110 001' return Sys_DC;     // CIVAC
        when '011 0111 1110 011' return Sys_DC;     // CIGVAC
        when '011 0111 1110 101' return Sys_DC;     // CIGDVAC
        when '100 0111 1110 000' return Sys_DC;     // CIPAE
        when '100 0111 1110 111' return Sys_DC;     // CIGDPAE
        when '110 0111 1110 001' return Sys_DC;     // CIPAPA
        when '110 0111 1110 101' return Sys_DC;     // CIGDPAPA
        when '000 0111 0001 000' return Sys_IC;     // IALLUIS
        when '000 0111 0101 000' return Sys_IC;     // IALLU
        when '011 0111 0101 001' return Sys_IC;     // IVAU
        when '000 1000 0001 000' return Sys_TLBI;   // VMALLE1OS
        when '000 1000 0001 001' return Sys_TLBI;   // VAE1OS
        when '000 1000 0001 010' return Sys_TLBI;   // ASIDE1OS
        when '000 1000 0001 011' return Sys_TLBI;   // VAAE1OS
        when '000 1000 0001 101' return Sys_TLBI;   // VALE1OS
        when '000 1000 0001 111' return Sys_TLBI;   // VAALE1OS
        when '000 1000 0010 001' return Sys_TLBI;   // RVAE1IS
        when '000 1000 0010 011' return Sys_TLBI;   // RVAAE1IS
        when '000 1000 0010 101' return Sys_TLBI;   // RVALE1IS
        when '000 1000 0010 111' return Sys_TLBI;   // RVAALE1IS
        when '000 1000 0011 000' return Sys_TLBI;   // VMALLE1IS
        when '000 1000 0011 001' return Sys_TLBI;   // VAE1IS
        when '000 1000 0011 010' return Sys_TLBI;   // ASIDE1IS
        when '000 1000 0011 011' return Sys_TLBI;   // VAAE1IS
```

```
        when '000 1000 0011 101' return Sys_TLBI;  // VALE1IS
        when '000 1000 0011 111' return Sys_TLBI;  // VAALE1IS
        when '000 1000 0101 001' return Sys_TLBI;  // RVAE1OS
        when '000 1000 0101 011' return Sys_TLBI;  // RVAAE1OS
        when '000 1000 0101 101' return Sys_TLBI;  // RVALE1OS
        when '000 1000 0101 111' return Sys_TLBI;  // RVAALE1OS
        when '000 1000 0110 001' return Sys_TLBI;  // RVAE1
        when '000 1000 0110 011' return Sys_TLBI;  // RVAAE1
        when '000 1000 0110 101' return Sys_TLBI;  // RVALE1
        when '000 1000 0110 111' return Sys_TLBI;  // RVAALE1
        when '000 1000 0111 000' return Sys_TLBI;  // VMALLE1
        when '000 1000 0111 001' return Sys_TLBI;  // VAE1
        when '000 1000 0111 010' return Sys_TLBI;  // ASIDE1
        when '000 1000 0111 011' return Sys_TLBI;  // VAAE1
        when '000 1000 0111 101' return Sys_TLBI;  // VALE1
        when '000 1000 0111 111' return Sys_TLBI;  // VAALE1
        when '000 1001 0001 000' return Sys_TLBI;  // VMALLE1OSNXS
        when '000 1001 0001 001' return Sys_TLBI;  // VAE1OSNXS
        when '000 1001 0001 010' return Sys_TLBI;  // ASIDE1OSNXS
        when '000 1001 0001 011' return Sys_TLBI;  // VAAE1OSNXS
        when '000 1001 0001 101' return Sys_TLBI;  // VALE1OSNXS
        when '000 1001 0001 111' return Sys_TLBI;  // VAALE1OSNXS
        when '000 1001 0010 001' return Sys_TLBI;  // RVAE1ISNXS
        when '000 1001 0010 011' return Sys_TLBI;  // RVAAE1ISNXS
        when '000 1001 0010 101' return Sys_TLBI;  // RVALE1ISNXS
        when '000 1001 0010 111' return Sys_TLBI;  // RVAALE1ISNXS
        when '000 1001 0011 000' return Sys_TLBI;  // VMALLE1ISNXS
        when '000 1001 0011 001' return Sys_TLBI;  // VAE1ISNXS
        when '000 1001 0011 010' return Sys_TLBI;  // ASIDE1ISNXS
        when '000 1001 0011 011' return Sys_TLBI;  // VAAE1ISNXS
        when '000 1001 0011 101' return Sys_TLBI;  // VALE1ISNXS
        when '000 1001 0011 111' return Sys_TLBI;  // VAALE1ISNXS
        when '000 1001 0101 001' return Sys_TLBI;  // RVAE1OSNXS
        when '000 1001 0101 011' return Sys_TLBI;  // RVAAE1OSNXS
        when '000 1001 0101 101' return Sys_TLBI;  // RVALE1OSNXS
        when '000 1001 0101 111' return Sys_TLBI;  // RVAALE1OSNXS
        when '000 1001 0110 001' return Sys_TLBI;  // RVAE1NXS
        when '000 1001 0110 011' return Sys_TLBI;  // RVAAE1NXS
        when '000 1001 0110 101' return Sys_TLBI;  // RVALE1NXS
        when '000 1001 0110 111' return Sys_TLBI;  // RVAALE1NXS
        when '000 1001 0111 000' return Sys_TLBI;  // VMALLE1NXS
        when '000 1001 0111 001' return Sys_TLBI;  // VAE1NXS
        when '000 1001 0111 010' return Sys_TLBI;  // ASIDE1NXS
        when '000 1001 0111 011' return Sys_TLBI;  // VAAE1NXS
        when '000 1001 0111 101' return Sys_TLBI;  // VALE1NXS
        when '000 1001 0111 111' return Sys_TLBI;  // VAALE1NXS
        when '100 1000 0000 001' return Sys_TLBI;  // IPAS2E1IS
        when '100 1000 0000 010' return Sys_TLBI;  // RIPAS2E1IS
        when '100 1000 0000 101' return Sys_TLBI;  // IPAS2LE1IS
        when '100 1000 0000 110' return Sys_TLBI;  // RIPAS2LE1IS
        when '100 1000 0001 000' return Sys_TLBI;  // ALLE2OS
        when '100 1000 0001 001' return Sys_TLBI;  // VAE2OS
        when '100 1000 0001 100' return Sys_TLBI;  // ALLE1OS
        when '100 1000 0001 101' return Sys_TLBI;  // VALE2OS
        when '100 1000 0001 110' return Sys_TLBI;  // VMALLS12E1OS
        when '100 1000 0010 001' return Sys_TLBI;  // RVAE2IS
        when '100 1000 0010 101' return Sys_TLBI;  // RVALE2IS
        when '100 1000 0011 000' return Sys_TLBI;  // ALLE2IS
        when '100 1000 0011 001' return Sys_TLBI;  // VAE2IS
        when '100 1000 0011 100' return Sys_TLBI;  // ALLE1IS
        when '100 1000 0011 101' return Sys_TLBI;  // VALE2IS
        when '100 1000 0011 110' return Sys_TLBI;  // VMALLS12E1IS
        when '100 1000 0100 000' return Sys_TLBI;  // IPAS2E1OS
        when '100 1000 0100 001' return Sys_TLBI;  // IPAS2E1
        when '100 1000 0100 010' return Sys_TLBI;  // RIPAS2E1
        when '100 1000 0100 011' return Sys_TLBI;  // RIPAS2E1OS
        when '100 1000 0100 100' return Sys_TLBI;  // IPAS2LE1OS
        when '100 1000 0100 101' return Sys_TLBI;  // IPAS2LE1
        when '100 1000 0100 110' return Sys_TLBI;  // RIPAS2LE1
        when '100 1000 0100 111' return Sys_TLBI;  // RIPAS2LE1OS
```

```
        when '100 1000 0101 001' return Sys_TLBI;  // RVAE2OS
        when '100 1000 0101 101' return Sys_TLBI;  // RVALE2OS
        when '100 1000 0110 001' return Sys_TLBI;  // RVAE2
        when '100 1000 0110 101' return Sys_TLBI;  // RVALE2
        when '100 1000 0111 000' return Sys_TLBI;  // ALLE2
        when '100 1000 0111 001' return Sys_TLBI;  // VAE2
        when '100 1000 0111 100' return Sys_TLBI;  // ALLE1
        when '100 1000 0111 101' return Sys_TLBI;  // VALE2
        when '100 1000 0111 110' return Sys_TLBI;  // VMALLS12E1
        when '100 1001 0000 001' return Sys_TLBI;  // IPAS2E1ISNXS
        when '100 1001 0000 010' return Sys_TLBI;  // RIPAS2E1ISNXS
        when '100 1001 0000 101' return Sys_TLBI;  // IPAS2LE1ISNXS
        when '100 1001 0000 110' return Sys_TLBI;  // RIPAS2LE1ISNXS
        when '100 1001 0001 000' return Sys_TLBI;  // ALLE2OSNXS
        when '100 1001 0001 001' return Sys_TLBI;  // VAE2OSNXS
        when '100 1001 0001 100' return Sys_TLBI;  // ALLE1OSNXS
        when '100 1001 0001 101' return Sys_TLBI;  // VALE2OSNXS
        when '100 1001 0001 110' return Sys_TLBI;  // VMALLS12E1OSNXS
        when '100 1001 0010 001' return Sys_TLBI;  // RVAE2ISNXS
        when '100 1001 0010 101' return Sys_TLBI;  // RVALE2ISNXS
        when '100 1001 0011 000' return Sys_TLBI;  // ALLE2ISNXS
        when '100 1001 0011 001' return Sys_TLBI;  // VAE2ISNXS
        when '100 1001 0011 100' return Sys_TLBI;  // ALLE1ISNXS
        when '100 1001 0011 101' return Sys_TLBI;  // VALE2ISNXS
        when '100 1001 0011 110' return Sys_TLBI;  // VMALLS12E1ISNXS
        when '100 1001 0100 000' return Sys_TLBI;  // IPAS2E1OSNXS
        when '100 1001 0100 001' return Sys_TLBI;  // IPAS2E1NXS
        when '100 1001 0100 010' return Sys_TLBI;  // RIPAS2E1NXS
        when '100 1001 0100 011' return Sys_TLBI;  // RIPAS2E1OSNXS
        when '100 1001 0100 100' return Sys_TLBI;  // IPAS2LE1OSNXS
        when '100 1001 0100 101' return Sys_TLBI;  // IPAS2LE1NXS
        when '100 1001 0100 110' return Sys_TLBI;  // RIPAS2LE1NXS
        when '100 1001 0100 111' return Sys_TLBI;  // RIPAS2LE1OSNXS
        when '100 1001 0101 001' return Sys_TLBI;  // RVAE2OSNXS
        when '100 1001 0101 101' return Sys_TLBI;  // RVALE2OSNXS
        when '100 1001 0110 001' return Sys_TLBI;  // RVAE2NXS
        when '100 1001 0110 101' return Sys_TLBI;  // RVALE2NXS
        when '100 1001 0111 000' return Sys_TLBI;  // ALLE2NXS
        when '100 1001 0111 001' return Sys_TLBI;  // VAE2NXS
        when '100 1001 0111 100' return Sys_TLBI;  // ALLE1NXS
        when '100 1001 0111 101' return Sys_TLBI;  // VALE2NXS
        when '100 1001 0111 110' return Sys_TLBI;  // VMALLS12E1NXS
        when '110 1000 0001 000' return Sys_TLBI;  // ALLE3OS
        when '110 1000 0001 001' return Sys_TLBI;  // VAE3OS
        when '110 1000 0001 100' return Sys_TLBI;  // PAALLOS
        when '110 1000 0001 101' return Sys_TLBI;  // VALE3OS
        when '110 1000 0010 001' return Sys_TLBI;  // RVAE3IS
        when '110 1000 0010 101' return Sys_TLBI;  // RVALE3IS
        when '110 1000 0011 000' return Sys_TLBI;  // ALLE3IS
        when '110 1000 0011 001' return Sys_TLBI;  // VAE3IS
        when '110 1000 0011 101' return Sys_TLBI;  // VALE3IS
        when '110 1000 0100 011' return Sys_TLBI;  // RPAOS
        when '110 1000 0100 111' return Sys_TLBI;  // RPALOS
        when '110 1000 0101 001' return Sys_TLBI;  // RVAE3OS
        when '110 1000 0101 101' return Sys_TLBI;  // RVALE3OS
        when '110 1000 0110 001' return Sys_TLBI;  // RVAE3
        when '110 1000 0110 101' return Sys_TLBI;  // RVALE3
        when '110 1000 0111 000' return Sys_TLBI;  // ALLE3
        when '110 1000 0111 001' return Sys_TLBI;  // VAE3
        when '110 1000 0111 100' return Sys_TLBI;  // PAALL
        when '110 1000 0111 101' return Sys_TLBI;  // VALE3
        when '110 1001 0001 000' return Sys_TLBI;  // ALLE3OSNXS
        when '110 1001 0001 001' return Sys_TLBI;  // VAE3OSNXS
        when '110 1001 0001 101' return Sys_TLBI;  // VALE3OSNXS
        when '110 1001 0010 001' return Sys_TLBI;  // RVAE3ISNXS
        when '110 1001 0010 101' return Sys_TLBI;  // RVALE3ISNXS
        when '110 1001 0011 000' return Sys_TLBI;  // ALLE3ISNXS
        when '110 1001 0011 001' return Sys_TLBI;  // VAE3ISNXS
        when '110 1001 0011 101' return Sys_TLBI;  // VALE3ISNXS
        when '110 1001 0101 001' return Sys_TLBI;  // RVAE3OSNXS
```

```
        when '110 1001 0101 101' return Sys_TLBI;  // RVALE3OSNXS
        when '110 1001 0110 001' return Sys_TLBI;  // RVAE3NXS
        when '110 1001 0110 101' return Sys_TLBI;  // RVALE3NXS
        when '110 1001 0111 000' return Sys_TLBI;  // ALLE3NXS
        when '110 1001 0111 001' return Sys_TLBI;  // VAE3NXS
        when '110 1001 0111 101' return Sys_TLBI;  // VALE3NXS
        otherwise                return Sys_SYS;
```

Library pseudocode for aarch64/functions/sysop/SystemOp

```
// SystemOp
// ========
// System instruction types.

enumeration SystemOp {Sys_AT, Sys_BRB, Sys_DC, Sys_IC, Sys_TLBI, Sys_SYS};
```

Library pseudocode for aarch64/functions/sysop_128/SysOp128

```
// SysOp128()
// ==========

SystemOp128 SysOp128(bits(3) op1, bits(4) CRn, bits(4) CRm, bits(3) op2)
    case op1:CRn:CRm:op2 of
        when '000 1000 0001 001' return Sys_TLBIP;  // VAE1OS
        when '000 1000 0001 011' return Sys_TLBIP;  // VAAE1OS
        when '000 1000 0001 101' return Sys_TLBIP;  // VALE1OS
        when '000 1000 0001 111' return Sys_TLBIP;  // VAALE1OS
        when '000 1000 0011 001' return Sys_TLBIP;  // VAE1IS
        when '000 1000 0011 011' return Sys_TLBIP;  // VAAE1IS
        when '000 1000 0011 101' return Sys_TLBIP;  // VALE1IS
        when '000 1000 0011 111' return Sys_TLBIP;  // VAALE1IS
        when '000 1000 0111 001' return Sys_TLBIP;  // VAE1
        when '000 1000 0111 011' return Sys_TLBIP;  // VAAE1
        when '000 1000 0111 101' return Sys_TLBIP;  // VALE1
        when '000 1000 0111 111' return Sys_TLBIP;  // VAALE1
        when '000 1001 0001 001' return Sys_TLBIP;  // VAE1OSNXS
        when '000 1001 0001 011' return Sys_TLBIP;  // VAAE1OSNXS
        when '000 1001 0001 101' return Sys_TLBIP;  // VALE1OSNXS
        when '000 1001 0001 111' return Sys_TLBIP;  // VAALE1OSNXS
        when '000 1001 0011 001' return Sys_TLBIP;  // VAE1ISNXS
        when '000 1001 0011 011' return Sys_TLBIP;  // VAAE1ISNXS
        when '000 1001 0011 101' return Sys_TLBIP;  // VALE1ISNXS
        when '000 1001 0011 111' return Sys_TLBIP;  // VAALE1ISNXS
        when '000 1001 0111 001' return Sys_TLBIP;  // VAE1NXS
        when '000 1001 0111 011' return Sys_TLBIP;  // VAAE1NXS
        when '000 1001 0111 101' return Sys_TLBIP;  // VALE1NXS
        when '000 1001 0111 111' return Sys_TLBIP;  // VAALE1NXS
        when '100 1000 0001 001' return Sys_TLBIP;  // VAE2OS
        when '100 1000 0001 101' return Sys_TLBIP;  // VALE2OS
        when '100 1000 0011 001' return Sys_TLBIP;  // VAE2IS
        when '100 1000 0011 101' return Sys_TLBIP;  // VALE2IS
        when '100 1000 0111 001' return Sys_TLBIP;  // VAE2
        when '100 1000 0111 101' return Sys_TLBIP;  // VALE2
        when '100 1001 0001 001' return Sys_TLBIP;  // VAE2OSNXS
        when '100 1001 0001 101' return Sys_TLBIP;  // VALE2OSNXS
        when '100 1001 0011 001' return Sys_TLBIP;  // VAE2ISNXS
        when '100 1001 0011 101' return Sys_TLBIP;  // VALE2ISNXS
        when '100 1001 0111 001' return Sys_TLBIP;  // VAE2NXS
        when '100 1001 0111 101' return Sys_TLBIP;  // VALE2NXS
        when '110 1000 0001 001' return Sys_TLBIP;  // VAE3OS
        when '110 1000 0001 101' return Sys_TLBIP;  // VALE3OS
        when '110 1000 0011 001' return Sys_TLBIP;  // VAE3IS
        when '110 1000 0011 101' return Sys_TLBIP;  // VALE3IS
        when '110 1000 0111 001' return Sys_TLBIP;  // VAE3
        when '110 1000 0111 101' return Sys_TLBIP;  // VALE3
        when '110 1001 0001 001' return Sys_TLBIP;  // VAE3OSNXS
        when '110 1001 0001 101' return Sys_TLBIP;  // VALE3OSNXS
        when '110 1001 0011 001' return Sys_TLBIP;  // VAE3ISNXS
        when '110 1001 0011 101' return Sys_TLBIP;  // VALE3ISNXS
        when '110 1001 0111 001' return Sys_TLBIP;  // VAE3NXS
        when '110 1001 0111 101' return Sys_TLBIP;  // VALE3NXS
        when '100 1000 0000 001' return Sys_TLBIP;  // IPAS2E1IS
        when '100 1000 0000 101' return Sys_TLBIP;  // IPAS2LE1IS
        when '100 1000 0100 000' return Sys_TLBIP;  // IPAS2E1OS
        when '100 1000 0100 001' return Sys_TLBIP;  // IPAS2E1
        when '100 1000 0100 100' return Sys_TLBIP;  // IPAS2LE1OS
        when '100 1000 0100 101' return Sys_TLBIP;  // IPAS2LE1
        when '100 1001 0000 001' return Sys_TLBIP;  // IPAS2E1ISNXS
        when '100 1001 0000 101' return Sys_TLBIP;  // IPAS2LE1ISNXS
        when '100 1001 0100 000' return Sys_TLBIP;  // IPAS2E1OSNXS
        when '100 1001 0100 001' return Sys_TLBIP;  // IPAS2E1NXS
        when '100 1001 0100 100' return Sys_TLBIP;  // IPAS2LE1OSNXS
        when '100 1001 0100 101' return Sys_TLBIP;  // IPAS2LE1NXS
        when '000 1000 0010 001' return Sys_TLBIP;  // RVAE1IS
        when '000 1000 0010 011' return Sys_TLBIP;  // RVAAE1IS
        when '000 1000 0010 101' return Sys_TLBIP;  // RVALE1IS
        when '000 1000 0010 111' return Sys_TLBIP;  // RVAALE1IS
        when '000 1000 0101 001' return Sys_TLBIP;  // RVAE1OS
```

```
        when '000 1000 0101 011' return Sys_TLBIP;  // RVAAE1OS
        when '000 1000 0101 101' return Sys_TLBIP;  // RVALE1OS
        when '000 1000 0101 111' return Sys_TLBIP;  // RVAALE1OS
        when '000 1000 0110 001' return Sys_TLBIP;  // RVAE1
        when '000 1000 0110 011' return Sys_TLBIP;  // RVAAE1
        when '000 1000 0110 101' return Sys_TLBIP;  // RVALE1
        when '000 1000 0110 111' return Sys_TLBIP;  // RVAALE1
        when '000 1001 0010 001' return Sys_TLBIP;  // RVAE1ISNXS
        when '000 1001 0010 011' return Sys_TLBIP;  // RVAAE1ISNXS
        when '000 1001 0010 101' return Sys_TLBIP;  // RVALE1ISNXS
        when '000 1001 0010 111' return Sys_TLBIP;  // RVAALE1ISNXS
        when '000 1001 0101 001' return Sys_TLBIP;  // RVAE1OSNXS
        when '000 1001 0101 011' return Sys_TLBIP;  // RVAAE1OSNXS
        when '000 1001 0101 101' return Sys_TLBIP;  // RVALE1OSNXS
        when '000 1001 0101 111' return Sys_TLBIP;  // RVAALE1OSNXS
        when '000 1001 0110 001' return Sys_TLBIP;  // RVAE1NXS
        when '000 1001 0110 011' return Sys_TLBIP;  // RVAAE1NXS
        when '000 1001 0110 101' return Sys_TLBIP;  // RVALE1NXS
        when '000 1001 0110 111' return Sys_TLBIP;  // RVAALE1NXS
        when '100 1000 0010 001' return Sys_TLBIP;  // RVAE2IS
        when '100 1000 0010 101' return Sys_TLBIP;  // RVALE2IS
        when '100 1000 0101 001' return Sys_TLBIP;  // RVAE2OS
        when '100 1000 0101 101' return Sys_TLBIP;  // RVALE2OS
        when '100 1000 0110 001' return Sys_TLBIP;  // RVAE2
        when '100 1000 0110 101' return Sys_TLBIP;  // RVALE2
        when '100 1001 0010 001' return Sys_TLBIP;  // RVAE2ISNXS
        when '100 1001 0010 101' return Sys_TLBIP;  // RVALE2ISNXS
        when '100 1001 0101 001' return Sys_TLBIP;  // RVAE2OSNXS
        when '100 1001 0101 101' return Sys_TLBIP;  // RVALE2OSNXS
        when '100 1001 0110 001' return Sys_TLBIP;  // RVAE2NXS
        when '100 1001 0110 101' return Sys_TLBIP;  // RVALE2NXS
        when '110 1000 0010 001' return Sys_TLBIP;  // RVAE3IS
        when '110 1000 0010 101' return Sys_TLBIP;  // RVALE3IS
        when '110 1000 0101 001' return Sys_TLBIP;  // RVAE3OS
        when '110 1000 0101 101' return Sys_TLBIP;  // RVALE3OS
        when '110 1000 0110 001' return Sys_TLBIP;  // RVAE3
        when '110 1000 0110 101' return Sys_TLBIP;  // RVALE3
        when '110 1001 0010 001' return Sys_TLBIP;  // RVAE3ISNXS
        when '110 1001 0010 101' return Sys_TLBIP;  // RVALE3ISNXS
        when '110 1001 0101 001' return Sys_TLBIP;  // RVAE3OSNXS
        when '110 1001 0101 101' return Sys_TLBIP;  // RVALE3OSNXS
        when '110 1001 0110 001' return Sys_TLBIP;  // RVAE3NXS
        when '110 1001 0110 101' return Sys_TLBIP;  // RVALE3NXS
        when '100 1000 0000 010' return Sys_TLBIP;  // RIPAS2E1IS
        when '100 1000 0000 110' return Sys_TLBIP;  // RIPAS2LE1IS
        when '100 1000 0100 010' return Sys_TLBIP;  // RIPAS2E1
        when '100 1000 0100 011' return Sys_TLBIP;  // RIPAS2E1OS
        when '100 1000 0100 110' return Sys_TLBIP;  // RIPAS2LE1
        when '100 1000 0100 111' return Sys_TLBIP;  // RIPAS2LE1OS
        when '100 1001 0000 010' return Sys_TLBIP;  // RIPAS2E1ISNXS
        when '100 1001 0000 110' return Sys_TLBIP;  // RIPAS2LE1ISNXS
        when '100 1001 0100 010' return Sys_TLBIP;  // RIPAS2E1NXS
        when '100 1001 0100 011' return Sys_TLBIP;  // RIPAS2E1OSNXS
        when '100 1001 0100 110' return Sys_TLBIP;  // RIPAS2LE1NXS
        when '100 1001 0100 111' return Sys_TLBIP;  // RIPAS2LE1OSNXS
        otherwise                return Sys_SYSP;
```

Library pseudocode for aarch64/functions/sysop_128/SystemOp128

```
// SystemOp128()
// =============
// System instruction types.

enumeration SystemOp128 {Sys_TLBIP, Sys_SYSP};
```

Library pseudocode for aarch64/functions/sysregisters/CNTKCTL

```
// CNTKCTL[] - non-assignment form
// ===============================

CNTKCTLType CNTKCTL[]
    bits(64) r;
    if IsInHost() then
        r = CNTHCTL_EL2;
        return r;
    r = CNTKCTL_EL1;
    return r;
```

Library pseudocode for aarch64/functions/sysregisters/CNTKCTLType

```
type CNTKCTLType;
```

Library pseudocode for aarch64/functions/sysregisters/CPACR

```
// CPACR[] - non-assignment form
// =============================

CPACRType CPACR[]
    bits(64) r;
    if IsInHost() then
        r = CPTR_EL2;
        return r;
    r = CPACR_EL1;
    return r;
```

Library pseudocode for aarch64/functions/sysregisters/CPACRType

```
type CPACRType;
```

Library pseudocode for aarch64/functions/sysregisters/ELR

```
// ELR[] - non-assignment form
// ===========================

bits(64) ELR[bits(2) el]
    bits(64) r;
    case el of
        when EL1  r = ELR_EL1;
        when EL2  r = ELR_EL2;
        when EL3  r = ELR_EL3;
        otherwise Unreachable();
    return r;

// ELR[] - non-assignment form
// ===========================

bits(64) ELR[]
    assert PSTATE.EL != EL0;
    return ELR[PSTATE.EL];

// ELR[] - assignment form
// =======================

ELR[bits(2) el] = bits(64) value
    bits(64) r = value;
    case el of
        when EL1  ELR_EL1 = r;
        when EL2  ELR_EL2 = r;
        when EL3  ELR_EL3 = r;
        otherwise Unreachable();
    return;

// ELR[] - assignment form
// =======================

ELR[] = bits(64) value
    assert PSTATE.EL != EL0;
    ELR[PSTATE.EL] = value;
    return;
```

Library pseudocode for aarch64/functions/sysregisters/ESR

```
// ESR[] - non-assignment form
// ===========================

ESRType ESR[bits(2) regime]
    bits(64) r;
    case regime of
        when EL1  r = ESR_EL1;
        when EL2  r = ESR_EL2;
        when EL3  r = ESR_EL3;
        otherwise Unreachable();
    return r;

// ESR[] - non-assignment form
// ===========================

ESRType ESR[]
    return ESR[S1TranslationRegime()];

// ESR[] - assignment form
// =======================

ESR[bits(2) regime] = ESRType value
    bits(64) r = value;
    case regime of
        when EL1  ESR_EL1 = r;
        when EL2  ESR_EL2 = r;
        when EL3  ESR_EL3 = r;
        otherwise Unreachable();
    return;

// ESR[] - assignment form
// =======================

ESR[] = ESRType value
    ESR[S1TranslationRegime()] = value;
```

Library pseudocode for aarch64/functions/sysregisters/ESRType

```
type ESRType;
```

Library pseudocode for aarch64/functions/sysregisters/FAR

```
// FAR[] - non-assignment form
// ===========================

bits(64) FAR[bits(2) regime]
    bits(64) r;
    case regime of
        when EL1  r = FAR_EL1;
        when EL2  r = FAR_EL2;
        when EL3  r = FAR_EL3;
        otherwise Unreachable();
    return r;

// FAR[] - non-assignment form
// ===========================

bits(64) FAR[]
    return FAR[S1TranslationRegime()];

// FAR[] - assignment form
// =======================

FAR[bits(2) regime] = bits(64) value
    bits(64) r = value;
    case regime of
        when EL1  FAR_EL1 = r;
        when EL2  FAR_EL2 = r;
        when EL3  FAR_EL3 = r;
        otherwise Unreachable();
    return;

// FAR[] - assignment form
// =======================

FAR[] = bits(64) value
    FAR[S1TranslationRegime()] = value;
    return;
```

Library pseudocode for aarch64/functions/sysregisters/PFAR

```
// PFAR[] - non-assignment form
// ============================

bits(64) PFAR[bits(2) regime]
    assert (HavePFAR() || (regime == EL3 && HaveRME()));
    bits(64) r;
    case regime of
        when EL1 r = PFAR_EL1;
        when EL2 r = PFAR_EL2;
        when EL3 r = MFAR_EL3;
        otherwise Unreachable();
    return r;

// PFAR[] - non-assignment form
// ============================

bits(64) PFAR[]
    return PFAR[S1TranslationRegime()];

// PFAR[] - assignment form
// ========================

PFAR[bits(2) regime] = bits(64) value
    bits(64) r = value;
    assert (HavePFAR() || (HaveRME() && regime == EL3));
    case regime of
        when EL1 PFAR_EL1 = r;
        when EL2 PFAR_EL2 = r;
        when EL3 MFAR_EL3 = r;
        otherwise Unreachable();
    return;

// PFAR[] - assignment form
// ========================

PFAR[] = bits(64) value
    PFAR[S1TranslationRegime()] = value;
    return;
```

Library pseudocode for aarch64/functions/sysregisters/S1PIRType

```
type S1PIRType;
```

Library pseudocode for aarch64/functions/sysregisters/S1PORType

```
type S1PORType;
```

Library pseudocode for aarch64/functions/sysregisters/S2PIRType

```
type S2PIRType;
```

Library pseudocode for aarch64/functions/sysregisters/S2PORType

```
type S2PORType;
```

Library pseudocode for aarch64/functions/sysregisters/SCTLR

```
// SCTLR[] - non-assignment form
// =============================

SCTLRType SCTLR[bits(2) regime]
    bits(64) r;
    case regime of
        when EL1  r = SCTLR_EL1;
        when EL2  r = SCTLR_EL2;
        when EL3  r = SCTLR_EL3;
        otherwise Unreachable();
    return r;

// SCTLR[] - non-assignment form
// =============================

SCTLRType SCTLR[]
    return SCTLR[S1TranslationRegime()];
```

Library pseudocode for aarch64/functions/sysregisters/SCTLRType

```
type SCTLRType;
```

Library pseudocode for aarch64/functions/sysregisters/VBAR

```
// VBAR[] - non-assignment form
// ============================

bits(64) VBAR[bits(2) regime]
    bits(64) r;
    case regime of
        when EL1  r = VBAR_EL1;
        when EL2  r = VBAR_EL2;
        when EL3  r = VBAR_EL3;
        otherwise Unreachable();
    return r;

// VBAR[] - non-assignment form
// ============================

bits(64) VBAR[]
    return VBAR[S1TranslationRegime()];
```

Library pseudocode for aarch64/functions/system/AArch64.AllocationTagAccessIsEnabled

```
// AArch64.AllocationTagAccessIsEnabled()
// ======================================
// Check whether access to Allocation Tags is enabled.

boolean AArch64.AllocationTagAccessIsEnabled(bits(2) el)
    if SCR_EL3.ATA == '0' && el IN {EL0, EL1, EL2} then
        return FALSE;
    if HCR_EL2.ATA == '0' && el IN {EL0, EL1} && EL2Enabled() && HCR_EL2.<E2H,TGE> != '11' then
        return FALSE;

    Regime regime = TranslationRegime(el);
    case regime of
        when Regime_EL3  return SCTLR_EL3.ATA == '1';
        when Regime_EL2  return SCTLR_EL2.ATA == '1';
        when Regime_EL20 return if el == EL0 then SCTLR_EL2.ATA0 == '1' else SCTLR_EL2.ATA == '1';
        when Regime_EL10 return if el == EL0 then SCTLR_EL1.ATA0 == '1' else SCTLR_EL1.ATA == '1';
        otherwise Unreachable();
```

Library pseudocode for aarch64/functions/system/AArch64.CheckSystemAccess

```
// AArch64.CheckSystemAccess()
// ===========================

AArch64.CheckSystemAccess(bits(2) op0, bits(3) op1, bits(4) crn,
                          bits(4) crm, bits(3) op2, bits(5) rt, bit read)
    if HaveBTIExt() then
        BranchTargetCheck();

    if (HaveTME() && TSTATE.depth > 0 &&
          !CheckTransactionalSystemAccess(op0, op1, crn, crm, op2, read)) then
        FailTransaction(TMFailure_ERR, FALSE);

    return;
```

Library pseudocode for aarch64/functions/system/AArch64.ChooseNonExcludedTag

```
// AArch64.ChooseNonExcludedTag()
// ==============================
// Return a tag derived from the start and the offset values, excluding
// any tags in the given mask.

bits(4) AArch64.ChooseNonExcludedTag(bits(4) tag_in, bits(4) offset_in, bits(16) exclude)
    bits(4) tag = tag_in;
    bits(4) offset = offset_in;

    if IsOnes(exclude) then
        return '0000';

    if offset == '0000' then
        while exclude<UInt(tag)> == '1' do
            tag = tag + '0001';

    while offset != '0000' do
        offset = offset - '0001';
        tag = tag + '0001';
        while exclude<UInt(tag)> == '1' do
            tag = tag + '0001';

    return tag;
```

Library pseudocode for aarch64/functions/system/AArch64.ExecutingBROrBLROrRetInstr

```
// AArch64.ExecutingBROrBLROrRetInstr()
// ====================================
// Returns TRUE if current instruction is a BR, BLR, RET, B[L]RA[B][Z], or RETA[B].

boolean AArch64.ExecutingBROrBLROrRetInstr()
    if !HaveBTIExt() then return FALSE;

    instr = ThisInstr();
    if instr<31:25> == '1101011' && instr<20:16> == '11111' then
        opc = instr<24:21>;
        return opc != '0101';
    else
        return FALSE;
```

Library pseudocode for aarch64/functions/system/AArch64.ExecutingBTIInstr

```
// AArch64.ExecutingBTIInstr()
// ===========================
// Returns TRUE if current instruction is a BTI.

boolean AArch64.ExecutingBTIInstr()
    if !HaveBTIExt() then return FALSE;

    instr = ThisInstr();
    if instr<31:22> == '1101010100' && instr<21:12> == '0000110010' && instr<4:0> == '11111' then
        CRm  = instr<11:8>;
        op2  = instr<7:5>;
        return (CRm == '0100' && op2<0> == '0');
    else
        return FALSE;
```

Library pseudocode for aarch64/functions/system/AArch64.ExecutingERETInstr

```
// AArch64.ExecutingERETInstr()
// ============================
// Returns TRUE if current instruction is ERET.

boolean AArch64.ExecutingERETInstr()
    instr = ThisInstr();
    return instr<31:12> == '11010110100111110000';
```

Library pseudocode for aarch64/functions/system/AArch64.ImpDefSysInstr

```
// AArch64.ImpDefSysInstr()
// ========================
// Execute an implementation-defined system instruction with write (source operand).

AArch64.ImpDefSysInstr(integer el, bits(3) op1, bits(4) CRn, bits(4) CRm, bits(3) op2, integer t);
```

Library pseudocode for aarch64/functions/system/AArch64.ImpDefSysInstr128

```
// AArch64.ImpDefSysInstr128()
// ===========================
// Execute an implementation-defined system instruction with write (128-bit source operand).

AArch64.ImpDefSysInstr128(integer el, bits(3) op1, bits(4) CRn,
                          bits(4) CRm, bits(3) op2,
                          integer t, integer t2);
```

Library pseudocode for aarch64/functions/system/AArch64.ImpDefSysInstrWithResult

```
// AArch64.ImpDefSysInstrWithResult()
// ==================================
// Execute an implementation-defined system instruction with read (result operand).

AArch64.ImpDefSysInstrWithResult(integer el, bits(3) op1, bits(4) CRn, bits(4) CRm, bits(3) op2);
```

Library pseudocode for aarch64/functions/system/AArch64.ImpDefSysRegRead

```
// AArch64.ImpDefSysRegRead()
// ==========================
// Read from an implementation-defined System register and write the contents of the register
// to X[t].

AArch64.ImpDefSysRegRead(bits(2) op0, bits(3) op1, bits(4) CRn, bits(4) CRm, bits(3) op2,
                         integer t);
```

Library pseudocode for aarch64/functions/system/AArch64.ImpDefSysRegRead128

```
// AArch64.ImpDefSysRegRead128()
// =============================
// Read from an 128-bit implementation-defined System register
// and write the contents of the register to X[t], X[t+1].

AArch64.ImpDefSysRegRead128(bits(2) op0, bits(3) op1, bits(4) CRn,
                            bits(4) CRm, bits(3) op2,
                            integer t, integer t2);
```

Library pseudocode for aarch64/functions/system/AArch64.ImpDefSysRegWrite

```
// AArch64.ImpDefSysRegWrite()
// ===========================
// Write to an implementation-defined System register.

AArch64.ImpDefSysRegWrite(bits(2) op0, bits(3) op1, bits(4) CRn, bits(4) CRm, bits(3) op2,
                          integer t);
```

Library pseudocode for aarch64/functions/system/AArch64.ImpDefSysRegWrite128

```
// AArch64.ImpDefSysRegWrite128()
// ==============================
// Write the contents of X[t], X[t+1] to an 128-bit implementation-defined System register.

AArch64.ImpDefSysRegWrite128(bits(2) op0, bits(3) op1, bits(4) CRn,
                             bits(4) CRm, bits(3) op2,
                             integer t, integer t2);
```

Library pseudocode for aarch64/functions/system/AArch64.NextRandomTagBit

```
// AArch64.NextRandomTagBit()
// ==========================
// Generate a random bit suitable for generating a random Allocation Tag.

bit AArch64.NextRandomTagBit()
    assert GCR_EL1.RRND == '0';
    bits(16) lfsr = RGSR_EL1.SEED<15:0>;
    bit top = lfsr<5> EOR lfsr<3> EOR lfsr<2> EOR lfsr<0>;
    RGSR_EL1.SEED<15:0> = top:lfsr<15:1>;
    return top;
```

Library pseudocode for aarch64/functions/system/AArch64.RandomTag

```
// AArch64.RandomTag()
// ===================
// Generate a random Allocation Tag.

bits(4) AArch64.RandomTag()
    bits(4) tag;
    for i = 0 to 3
        tag<i> = AArch64.NextRandomTagBit();
    return tag;
```

Library pseudocode for aarch64/functions/system/AArch64.SysInstr

```
// AArch64.SysInstr()
// ==================
// Execute a system instruction with write (source operand).

AArch64.SysInstr(integer op0, integer op1, integer crn, integer crm, integer op2, integer t);
```

Library pseudocode for aarch64/functions/system/AArch64.SysInstrWithResult

```
// AArch64.SysInstrWithResult()
// ============================
// Execute a system instruction with read (result operand).
// Writes the result of the instruction to X[t].

AArch64.SysInstrWithResult(integer op0, integer op1, integer crn, integer crm, integer op2,
                           integer t);
```

Library pseudocode for aarch64/functions/system/AArch64.SysRegRead

```
// AArch64.SysRegRead()
// ====================
// Read from a System register and write the contents of the register to X[t].

AArch64.SysRegRead(integer op0, integer op1, integer crn, integer crm, integer op2, integer t);
```

Library pseudocode for aarch64/functions/system/AArch64.SysRegWrite

```
// AArch64.SysRegWrite()
// =====================
// Write to a System register.

AArch64.SysRegWrite(integer op0, integer op1, integer crn, integer crm, integer op2, integer t);
```

Library pseudocode for aarch64/functions/system/BTypeCompatible

```
boolean BTypeCompatible;
```

Library pseudocode for aarch64/functions/system/BTypeCompatible_BTI

```
// BTypeCompatible_BTI
// ===================
// This function determines whether a given hint encoding is compatible with the current value of
// PSTATE.BTYPE. A value of TRUE here indicates a valid Branch Target Identification instruction.

boolean BTypeCompatible_BTI(bits(2) hintcode)
    case hintcode of
        when '00'
            return FALSE;
        when '01'
            return PSTATE.BTYPE != '11';
        when '10'
            return PSTATE.BTYPE != '10';
        when '11'
            return TRUE;
```

Library pseudocode for aarch64/functions/system/BTypeCompatible_PACIXSP

```
// BTypeCompatible_PACIXSP()
// =========================
// Returns TRUE if PACIASP, PACIBSP instruction is implicit compatible with PSTATE.BTYPE,
// FALSE otherwise.

boolean BTypeCompatible_PACIXSP()
    if PSTATE.BTYPE IN {'01', '10'} then
        return TRUE;
    elsif PSTATE.BTYPE == '11' then
        index = if PSTATE.EL == EL0 then 35 else 36;
        return SCTLR[]<index> == '0';
    else
        return FALSE;
```

Library pseudocode for aarch64/functions/system/BTypeNext

```
bits(2) BTypeNext;
```

Library pseudocode for aarch64/functions/system/ChooseRandomNonExcludedTag

```
// ChooseRandomNonExcludedTag()
// ============================
// The ChooseRandomNonExcludedTag function is used when GCR_EL1.RRND == '1' to generate random
// Allocation Tags.
//
// The resulting Allocation Tag is selected from the set [0,15], excluding any Allocation Tag where
// exclude[tag_value] == 1. If 'exclude' is all Ones, the returned Allocation Tag is '0000'.
//
// This function is permitted to generate a non-deterministic selection from the set of non-excluded
// Allocation Tags. A reasonable implementation is described by the Pseudocode used when
// GCR_EL1.RRND is 0, but with a non-deterministic implementation of NextRandomTagBit().
// Implementations may choose to behave the same as GCR_EL1.RRND=0.
//
// This function can read RGSR_EL1 and/or write RGSR_EL1 to an IMPLEMENTATION DEFINED value.
// If it is not capable of writing RGSR_EL1.SEED[15:0] to zero from a previous nonzero
// RGSR_EL1.SEED value, it is IMPLEMENTATION DEFINED whether the randomness is significantly
// impacted if RGSR_EL1.SEED[15:0] is set to zero.

bits(4) ChooseRandomNonExcludedTag(bits(16) exclude_in);
```

Library pseudocode for aarch64/functions/system/InGuardedPage

```
boolean InGuardedPage;
```

Library pseudocode for aarch64/functions/system/IsHCRXEL2Enabled

```
// IsHCRXEL2Enabled()
// ==================
// Returns TRUE if access to HCRX_EL2 register is enabled, and FALSE otherwise.
// Indirect read of HCRX_EL2 returns 0 when access is not enabled.

boolean IsHCRXEL2Enabled()
    if !HaveFeatHCX() then return FALSE;
    if HaveEL(EL3) && SCR_EL3.HXEn == '0' then
        return FALSE;

    return EL2Enabled();
```

Library pseudocode for aarch64/functions/system/IsSCTLR2EL1Enabled

```
// IsSCTLR2EL1Enabled()
// ====================
// Returns TRUE if access to SCTLR2_EL1 register is enabled, and FALSE otherwise.
// Indirect read of SCTLR2_EL1 returns 0 when access is not enabled.

boolean IsSCTLR2EL1Enabled()
    if !HaveFeatSCTLR2() then return FALSE;
    if HaveEL(EL3) && SCR_EL3.SCTLR2En == '0' then
        return FALSE;
    elsif (EL2Enabled() && (!IsHCRXEL2Enabled() || HCRX_EL2.SCTLR2En == '0')) then
        return FALSE;
    else
        return TRUE;
```

Library pseudocode for aarch64/functions/system/IsSCTLR2EL2Enabled

```
// IsSCTLR2EL2Enabled()
// ====================
// Returns TRUE if access to SCTLR2_EL2 register is enabled, and FALSE otherwise.
// Indirect read of SCTLR2_EL2 returns 0 when access is not enabled.

boolean IsSCTLR2EL2Enabled()
    if !HaveFeatSCTLR2() then return FALSE;
    if HaveEL(EL3) && SCR_EL3.SCTLR2En == '0' then
        return FALSE;

    return EL2Enabled();
```

Library pseudocode for aarch64/functions/system/IsTCR2EL1Enabled

```
// IsTCR2EL1Enabled()
// ==================
// Returns TRUE if access to TCR2_EL1 register is enabled, and FALSE otherwise.
// Indirect read of TCR2_EL1 returns 0 when access is not enabled.

boolean IsTCR2EL1Enabled()
    if !HaveFeatTCR2() then return FALSE;
    if HaveEL(EL3) && SCR_EL3.TCR2En == '0' then
        return FALSE;
    elsif (EL2Enabled() && (!IsHCRXEL2Enabled() || HCRX_EL2.TCR2En == '0')) then
        return FALSE;
    else
        return TRUE;
```

Library pseudocode for aarch64/functions/system/IsTCR2EL2Enabled

```
// IsTCR2EL2Enabled()
// ==================
// Returns TRUE if access to TCR2_EL2 register is enabled, and FALSE otherwise.
// Indirect read of TCR2_EL2 returns 0 when access is not enabled.

boolean IsTCR2EL2Enabled()
    if !HaveFeatTCR2() then return FALSE;
    if HaveEL(EL3) && SCR_EL3.TCR2En == '0' then
        return FALSE;

    return EL2Enabled();
```

Library pseudocode for aarch64/functions/system/SetBTypeCompatible

```
// SetBTypeCompatible()
// ====================
// Sets the value of BTypeCompatible global variable used by BTI

SetBTypeCompatible(boolean x)
    BTypeCompatible = x;
```

Library pseudocode for aarch64/functions/system/SetBTypeNext

```
// SetBTypeNext()
// ==============
// Set the value of BTypeNext global variable used by BTI

SetBTypeNext(bits(2) x)
    BTypeNext = x;
```

Library pseudocode for aarch64/functions/system/SetInGuardedPage

```
// SetInGuardedPage()
// ==================
// Global state updated to denote if memory access is from a guarded page.

SetInGuardedPage(boolean guardedpage)
    InGuardedPage = guardedpage;
```

Library pseudocode for aarch64/functions/system128/AArch64.SysInstr128

```
// AArch64.SysInstr128()
// =====================
// Execute a system instruction with write (2 64-bit source operands).

AArch64.SysInstr128(integer op0, integer op1, integer crn, integer crm,
                    integer op2, integer t, integer t2);
```

Library pseudocode for aarch64/functions/system128/AArch64.SysRegRead128

```
// AArch64.SysRegRead128()
// =======================
// Read from a 128-bit System register and write the contents of the register to X[t] and X[t2].

AArch64.SysRegRead128(integer op0, integer op1, integer crn, integer crm,
                      integer op2, integer t, integer t2);
```

Library pseudocode for aarch64/functions/system128/AArch64.SysRegWrite128

```
// AArch64.SysRegWrite128()
// ========================
// Read the contents of X[t] and X[t2] and write the contents to a 128-bit System register.

AArch64.SysRegWrite128(integer op0, integer op1, integer crn, integer crm,
                       integer op2, integer t, integer t2);
```

Library pseudocode for aarch64/functions/tlbi/AArch64.TLBIP_IPAS2

```
// AArch64.TLBIP_IPAS2()
// =====================
// Invalidate by IPA all stage 2 only TLB entries in the indicated shareability
// domain matching the indicated VMID in the indicated regime with the indicated security state.
// Note: stage 1 and stage 2 combined entries are not in the scope of this operation.
// IPA and related parameters of the are derived from Xt.

AArch64.TLBIP_IPAS2(SecurityState security, Regime regime, bits(16) vmid,
                    Shareability shareability, TLBILevel level, TLBIMemAttr attr, bits(128) Xt)
    assert PSTATE.EL IN {EL3, EL2};

    TLBIRecord r;
    r.op           = TLBIOp_IPAS2;
    r.from_aarch64 = TRUE;
    r.security     = security;
    r.regime       = regime;
    r.vmid         = vmid;
    r.level        = level;
    r.attr         = attr;
    r.ttl          = Xt<47:44>;
    r.address      = ZeroExtend(Xt<107:64> : Zeros(12), 64);
    r.d64          = r.ttl IN {'00xx'};
    r.d128         = TRUE;

    case security of
        when SS_NonSecure
            r.ipaspace = PAS_NonSecure;
        when SS_Secure
            r.ipaspace = if Xt<63> == '1' then PAS_NonSecure else PAS_Secure;
        when SS_Realm
            r.ipaspace = PAS_Realm;
        otherwise
            // Root security state does not have stage 2 translation
            Unreachable();

    TLBI(r);
    if shareability != Shareability_NSH then Broadcast(shareability, r);
    return;
```

Library pseudocode for aarch64/functions/tlbi/AArch64.TLBIP_RIPAS2

```
// AArch64.TLBIP_RIPAS2()
// ======================
// Range invalidate by IPA all stage 2 only TLB entries in the indicated
// shareability domain matching the indicated VMID in the indicated regime with the indicated
// security state.
// Note: stage 1 and stage 2 combined entries are not in the scope of this operation.
// The range of IPA and related parameters of the are derived from Xt.

AArch64.TLBIP_RIPAS2(SecurityState security, Regime regime, bits(16) vmid,
                     Shareability shareability, TLBILevel level, TLBIMemAttr attr, bits(128) Xt)
    assert PSTATE.EL IN {EL3, EL2, EL1};

    TLBIRecord r;
    r.op           = TLBIOp_RIPAS2;
    r.from_aarch64 = TRUE;
    r.security     = security;
    r.regime       = regime;
    r.vmid         = vmid;
    r.level        = level;
    r.attr         = attr;
    r.ttl<1:0>     = Xt<38:37>;
    r.d64          = r.ttl<1:0> == '00';
    r.d128         = TRUE;

    bits(2) tg        = Xt<47:46>;
    integer scale     = UInt(Xt<45:44>);
    integer num       = UInt(Xt<43:39>);
    integer baseaddr  = SInt(Xt<36:0>);

    boolean valid;

    (valid, r.tg, r.address, r.end_address) = TLBIPRange(regime, Xt);

    if !valid then return;

    case security of
        when SS_NonSecure
            r.ipaspace = PAS_NonSecure;
        when SS_Secure
            r.ipaspace = if Xt<63> == '1' then PAS_NonSecure else PAS_Secure;
        when SS_Realm
            r.ipaspace = PAS_Realm;
        otherwise
            // Root security state does not have stage 2 translation
            Unreachable();

    TLBI(r);
    if shareability != Shareability_NSH then Broadcast(shareability, r);
    return;
```

Library pseudocode for aarch64/functions/tlbi/AArch64.TLBIP_RVA

```
// AArch64.TLBIP_RVA()
// ===================
// Range invalidate by VA range all stage 1 TLB entries in the indicated
// shareability domain matching the indicated VMID and ASID (where regime
// supports VMID, ASID) in the indicated regime with the indicated security state.
// ASID, and range related parameters are derived from Xt.
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.

AArch64.TLBIP_RVA(SecurityState security, Regime regime, bits(16) vmid,
                  Shareability shareability, TLBILevel level, TLBIMemAttr attr, bits(128)  Xt)
    assert PSTATE.EL IN {EL3, EL2, EL1};

    TLBIRecord r;
    r.op           = TLBIOp_RVA;
    r.from_aarch64 = TRUE;
    r.security     = security;
    r.regime       = regime;
    r.vmid         = vmid;
    r.level        = level;
    r.attr         = attr;
    r.asid         = Xt<63:48>;
    r.ttl<1:0>     = Xt<38:37>;
    r.d64          = r.ttl<1:0> == '00';
    r.d128         = TRUE;

    boolean valid;

    (valid, r.tg, r.address, r.end_address) = TLBIPRange(regime, Xt);

    if !valid then return;

    TLBI(r);
    if shareability != Shareability_NSH then Broadcast(shareability, r);
    return;
```

Library pseudocode for aarch64/functions/tlbi/AArch64.TLBIP_RVAA

```
// AArch64.TLBIP_RVAA()
// ====================
// Range invalidate by VA range all stage 1 TLB entries in the indicated
// shareability domain matching the indicated VMID (where regimesupports VMID)
// and all ASID in the indicated regime with the indicated security state.
// VA range related parameters are derived from Xt.
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.

AArch64.TLBIP_RVAA(SecurityState security, Regime  regime, bits(16) vmid,
                   Shareability shareability, TLBILevel level, TLBIMemAttr attr, bits(128) Xt)
    assert PSTATE.EL IN {EL3, EL2, EL1};

    TLBIRecord r;
    r.op           = TLBIOp_RVAA;
    r.from_aarch64 = TRUE;
    r.security     = security;
    r.regime       = regime;
    r.vmid         = vmid;
    r.level        = level;
    r.attr         = attr;
    r.ttl<1:0>     = Xt<38:37>;
    r.d64          = r.ttl<1:0> == '00';
    r.d128         = TRUE;

    bits(2) tg        = Xt<47:46>;
    integer scale     = UInt(Xt<45:44>);
    integer num       = UInt(Xt<43:39>);
    integer baseaddr  = SInt(Xt<36:0>);

    boolean valid;

    (valid, r.tg, r.address, r.end_address) = TLBIPRange(regime, Xt);

    if !valid then return;

    TLBI(r);
    if shareability != Shareability_NSH then Broadcast(shareability, r);
    return;
```

Library pseudocode for aarch64/functions/tlbi/AArch64.TLBIP_VA

```
// AArch64.TLBIP_VA()
// ==================
// Invalidate by VA all stage 1 TLB entries in the indicated shareability domain
// matching the indicated VMID and ASID (where regime supports VMID, ASID) in the indicated regime
// with the indicated security state.
// ASID, VA and related parameters are derived from Xt.
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.

AArch64.TLBIP_VA(SecurityState security, Regime regime, bits(16) vmid,
                 Shareability shareability, TLBILevel level,  TLBIMemAttr attr, bits(128) Xt)
    assert PSTATE.EL IN {EL3, EL2, EL1};

    TLBIRecord r;
    r.op           = TLBIOp_VA;
    r.from_aarch64 = TRUE;
    r.security     = security;
    r.regime       = regime;
    r.vmid         = vmid;
    r.level        = level;
    r.attr         = attr;
    r.asid         = Xt<63:48>;
    r.ttl          = Xt<47:44>;
    r.address      = ZeroExtend(Xt<107:64> : Zeros(12), 64);
    r.d64          = r.ttl IN {'00xx'};
    r.d128         = TRUE;

    TLBI(r);
    if shareability != Shareability_NSH then Broadcast(shareability, r);
    return;
```

Library pseudocode for aarch64/functions/tlbi/AArch64.TLBIP_VAA

```
// AArch64.TLBIP_VAA()
// ===================
// Invalidate by VA all stage 1 TLB entries in the indicated shareability domain
// matching the indicated VMID (where regime supports VMID) and all ASID in the indicated regime
// with the indicated security state.
// VA and related parameters are derived from Xt.
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.

AArch64.TLBIP_VAA(SecurityState security, Regime regime, bits(16) vmid,
                  Shareability shareability, TLBILevel level, TLBIMemAttr attr,  bits(128) Xt)
    assert PSTATE.EL IN {EL3, EL2, EL1};

    TLBIRecord r;
    r.op           = TLBIOp_VAA;
    r.from_aarch64 = TRUE;
    r.security     = security;
    r.regime       = regime;
    r.vmid         = vmid;
    r.level        = level;
    r.attr         = attr;
    r.ttl          = Xt<47:44>;
    r.address      = ZeroExtend(Xt<107:64> : Zeros(12), 64);
    r.d64          = r.ttl IN {'00xx'};
    r.d128         = TRUE;

    TLBI(r);
    if shareability != Shareability_NSH then Broadcast(shareability, r);
    return;
```

Library pseudocode for aarch64/functions/tlbi/AArch64.TLBI_ALL

```
// AArch64.TLBI_ALL()
// ==================
// Invalidate all entries for the indicated translation regime with the
// the indicated security state for all TLBs within the indicated shareability domain.
// Invalidation applies to all applicable stage 1 and stage 2 entries.

AArch64.TLBI_ALL(SecurityState security, Regime regime, Shareability shareability, TLBIMemAttr attr)
    assert PSTATE.EL IN {EL3, EL2};

    TLBIRecord r;
    r.op           = TLBIOp_ALL;
    r.from_aarch64 = TRUE;
    r.security     = security;
    r.regime       = regime;
    r.level        = TLBILevel_Any;
    r.attr         = attr;

    TLBI(r);
    if shareability != Shareability_NSH then Broadcast(shareability, r);
    return;
```

Library pseudocode for aarch64/functions/tlbi/AArch64.TLBI_ASID

```
// AArch64.TLBI_ASID()
// ===================
// Invalidate all stage 1 entries matching the indicated VMID (where regime supports)
// and ASID in the parameter Xt in the indicated translation regime with the
// indicated security state for all TLBs within the indicated shareability domain.
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.

AArch64.TLBI_ASID(SecurityState security, Regime regime, bits(16) vmid,
                  Shareability shareability, TLBIMemAttr attr, bits(64) Xt)
    assert PSTATE.EL IN {EL3, EL2, EL1};

    TLBIRecord r;
    r.op           = TLBIOp_ASID;
    r.from_aarch64 = TRUE;
    r.security     = security;
    r.regime       = regime;
    r.vmid         = vmid;
    r.level        = TLBILevel_Any;
    r.attr         = attr;
    r.asid         = Xt<63:48>;

    TLBI(r);
    if shareability != Shareability_NSH then Broadcast(shareability, r);
    return;
```

Library pseudocode for aarch64/functions/tlbi/AArch64.TLBI_IPAS2

```
// AArch64.TLBI_IPAS2()
// ====================
// Invalidate by IPA all stage 2 only TLB entries in the indicated shareability
// domain matching the indicated VMID in the indicated regime with the indicated security state.
// Note: stage 1 and stage 2 combined entries are not in the scope of this operation.
// IPA and related parameters of the are derived from Xt.

AArch64.TLBI_IPAS2(SecurityState security, Regime regime, bits(16) vmid,
                   Shareability shareability, TLBILevel level, TLBIMemAttr attr, bits(64) Xt)
    assert PSTATE.EL IN {EL3, EL2};

    TLBIRecord r;
    r.op           = TLBIOp_IPAS2;
    r.from_aarch64 = TRUE;
    r.security     = security;
    r.regime       = regime;
    r.vmid         = vmid;
    r.level        = level;
    r.attr         = attr;
    r.ttl          = Xt<47:44>;
    r.address      = ZeroExtend(Xt<39:0> : Zeros(12), 64);
    r.d64          = TRUE;
    r.d128         = r.ttl IN {'00xx'};

    case security of
        when SS_NonSecure
            r.ipaspace = PAS_NonSecure;
        when SS_Secure
            r.ipaspace = if Xt<63> == '1' then PAS_NonSecure else PAS_Secure;
        when SS_Realm
            r.ipaspace = PAS_Realm;
        otherwise
            // Root security state does not have stage 2 translation
            Unreachable();

    TLBI(r);
    if shareability != Shareability_NSH then Broadcast(shareability, r);
    return;
```

Library pseudocode for aarch64/functions/tlbi/AArch64.TLBI_PAALL

```
// AArch64.TLBI_PAALL()
// ====================
// TLB Invalidate ALL GPT Information.
// Invalidates cached copies of GPT entries from TLBs in the indicated
// Shareabilty domain.
// The invalidation applies to all TLB entries containing GPT information.

AArch64.TLBI_PAALL(Shareability shareability)
    assert HaveRME() && PSTATE.EL == EL3;

    TLBIRecord r;

    // r.security and r.regime do not apply for TLBI by PA operations
    r.op    = TLBIOp_PAALL;
    r.level = TLBILevel_Any;
    r.attr  = TLBI_AllAttr;

    TLBI(r);
    if shareability != Shareability_NSH then Broadcast(shareability, r);

    return;
```

Library pseudocode for aarch64/functions/tlbi/AArch64.TLBI_RIPAS2

```
// AArch64.TLBI_RIPAS2()
// =====================
// Range invalidate by IPA all stage 2 only TLB entries in the indicated
// shareability domain matching the indicated VMID in the indicated regime with the indicated
// security state.
// Note: stage 1 and stage 2 combined entries are not in the scope of this operation.
// The range of IPA and related parameters of the are derived from Xt.

AArch64.TLBI_RIPAS2(SecurityState security, Regime regime, bits(16) vmid,
                    Shareability shareability, TLBILevel level, TLBIMemAttr attr, bits(64) Xt)
    assert PSTATE.EL IN {EL3, EL2, EL1};

    TLBIRecord r;
    r.op           = TLBIOp_RIPAS2;
    r.from_aarch64 = TRUE;
    r.security     = security;
    r.regime       = regime;
    r.vmid         = vmid;
    r.level        = level;
    r.attr         = attr;
    r.ttl<1:0>     = Xt<38:37>;
    r.d64          = TRUE;
    r.d128         = r.ttl<1:0> == '00';

    bits(2) tg        = Xt<47:46>;
    integer scale     = UInt(Xt<45:44>);
    integer num       = UInt(Xt<43:39>);
    integer baseaddr  = SInt(Xt<36:0>);

    boolean valid;

    (valid, r.tg, r.address, r.end_address) = TLBIRange(regime, Xt);

    if !valid then return;

    case security of
        when SS_NonSecure
            r.ipaspace = PAS_NonSecure;
        when SS_Secure
            r.ipaspace = if Xt<63> == '1' then PAS_NonSecure else PAS_Secure;
        when SS_Realm
            r.ipaspace = PAS_Realm;
        otherwise
            // Root security state does not have stage 2 translation
            Unreachable();

    TLBI(r);
    if shareability != Shareability_NSH then Broadcast(shareability, r);
    return;
```

Library pseudocode for aarch64/functions/tlbi/AArch64.TLBI_RPA

```
// AArch64.TLBI_RPA()
// ==================
// TLB Range Invalidate GPT Information by PA.
// Invalidates cached copies of GPT entries from TLBs in the indicated
// Shareabilty domain.
// The invalidation applies to TLB entries containing GPT information relating
// to the indicated physical address range.
// When the indicated level is
//     TLBILevel_Any  : this applies to TLB entries containing GPT information
//                      from all levels of the GPT walk
//     TLBILevel_Last : this applies to TLB entries containing GPT information
//                      from the last level of the GPT walk

AArch64.TLBI_RPA(TLBILevel level, bits(64) Xt, Shareability shareability)
    assert HaveRME() && PSTATE.EL == EL3;

    TLBIRecord r;
    integer range_bits;
    integer p;

    // r.security and r.regime do not apply for TLBI by PA operations
    r.op    = TLBIOp_RPA;
    r.level = level;
    r.attr  = TLBI_AllAttr;

    // SIZE field
    case Xt<47:44> of
        when '0000' range_bits = 12; // 4KB
        when '0001' range_bits = 14; // 16KB
        when '0010' range_bits = 16; // 64KB
        when '0011' range_bits = 21; // 2MB
        when '0100' range_bits = 25; // 32MB
        when '0101' range_bits = 29; // 512MB
        when '0110' range_bits = 30; // 1GB
        when '0111' range_bits = 34; // 16GB
        when '1000' range_bits = 36; // 64GB
        when '1001' range_bits = 39; // 512GB
        otherwise   range_bits = 0;  // Reserved encoding

    // If SIZE selects a range smaller than PGS, then PGS is used instead
    case DecodePGS(GPCCR_EL3.PGS) of
        when PGS_4KB  p = 12;
        when PGS_16KB p = 14;
        when PGS_64KB p = 16;

    if range_bits < p then
        range_bits = p;

    bits(52) BaseADDR = Zeros(52);
    case GPCCR_EL3.PGS of
        when '00' BaseADDR<51:12> = Xt<39:0>;   // 4KB
        when '10' BaseADDR<51:14> = Xt<39:2>;   // 16KB
        when '01' BaseADDR<51:16> = Xt<39:4>;   // 64KB

    // The calculation here automatically aligns BaseADDR to the size of
    // the region specifed in SIZE. However, the architecture does not
    // require this alignment and if BaseADDR is not aligned to the region
    // specified by SIZE then no entries are required to be invalidated.
    bits(52) start_addr = BaseADDR AND NOT ZeroExtend(Ones(range_bits), 52);
    bits(52) end_addr   = start_addr + ZeroExtend(Ones(range_bits), 52);

    // PASpace is not considered in TLBI by PA operations
    r.address     = ZeroExtend(start_addr, 64);
    r.end_address = ZeroExtend(end_addr, 64);

    TLBI(r);
    if shareability != Shareability_NSH then Broadcast(shareability, r);
```

Library pseudocode for aarch64/functions/tlbi/AArch64.TLBI_RVA

```
// AArch64.TLBI_RVA()
// ==================
// Range invalidate by VA range all stage 1 TLB entries in the indicated
// shareability domain matching the indicated VMID and ASID (where regime
// supports VMID, ASID) in the indicated regime with the indicated security state.
// ASID, and range related parameters are derived from Xt.
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.

AArch64.TLBI_RVA(SecurityState security, Regime regime, bits(16) vmid,
                 Shareability shareability, TLBILevel level, TLBIMemAttr attr, bits(64)  Xt)
    assert PSTATE.EL IN {EL3, EL2, EL1};

    TLBIRecord r;
    r.op           = TLBIOp_RVA;
    r.from_aarch64 = TRUE;
    r.security     = security;
    r.regime       = regime;
    r.vmid         = vmid;
    r.level        = level;
    r.attr         = attr;
    r.asid         = Xt<63:48>;
    r.ttl<1:0>     = Xt<38:37>;
    r.d64          = TRUE;
    r.d128         = r.ttl<1:0> == '00';

    boolean valid;

    (valid, r.tg, r.address, r.end_address) = TLBIRange(regime, Xt);

    if !valid then return;

    TLBI(r);
    if shareability != Shareability_NSH then Broadcast(shareability, r);
    return;
```

Library pseudocode for aarch64/functions/tlbi/AArch64.TLBI_RVAA

```
// AArch64.TLBI_RVAA()
// ===================
// Range invalidate by VA range all stage 1 TLB entries in the indicated
// shareability domain matching the indicated VMID (where regimesupports VMID)
// and all ASID in the indicated regime with the indicated security state.
// VA range related parameters are derived from Xt.
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.

AArch64.TLBI_RVAA(SecurityState security, Regime  regime, bits(16) vmid,
                  Shareability shareability, TLBILevel level, TLBIMemAttr attr, bits(64) Xt)
    assert PSTATE.EL IN {EL3, EL2, EL1};

    TLBIRecord r;
    r.op           = TLBIOp_RVAA;
    r.from_aarch64 = TRUE;
    r.security     = security;
    r.regime       = regime;
    r.vmid         = vmid;
    r.level        = level;
    r.attr         = attr;
    r.ttl<1:0>     = Xt<38:37>;
    r.d64          = TRUE;
    r.d128         = r.ttl<1:0> == '00';

    bits(2) tg        = Xt<47:46>;
    integer scale     = UInt(Xt<45:44>);
    integer num       = UInt(Xt<43:39>);
    integer baseaddr  = SInt(Xt<36:0>);

    boolean valid;

    (valid, r.tg, r.address, r.end_address) = TLBIRange(regime, Xt);

    if !valid then return;

    TLBI(r);
    if shareability != Shareability_NSH then Broadcast(shareability, r);
    return;
```

Library pseudocode for aarch64/functions/tlbi/AArch64.TLBI_VA

```
// AArch64.TLBI_VA()
// =================
// Invalidate by VA all stage 1 TLB entries in the indicated shareability domain
// matching the indicated VMID and ASID (where regime supports VMID, ASID) in the indicated regime
// with the indicated security state.
// ASID, VA and related parameters are derived from Xt.
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.

AArch64.TLBI_VA(SecurityState security, Regime regime, bits(16) vmid,
                Shareability shareability, TLBILevel level,  TLBIMemAttr attr, bits(64) Xt)
    assert PSTATE.EL IN {EL3, EL2, EL1};

    TLBIRecord r;
    r.op           = TLBIOp_VA;
    r.from_aarch64 = TRUE;
    r.security     = security;
    r.regime       = regime;
    r.vmid         = vmid;
    r.level        = level;
    r.attr         = attr;
    r.asid         = Xt<63:48>;
    r.ttl          = Xt<47:44>;
    r.address      = ZeroExtend(Xt<43:0> : Zeros(12), 64);
    r.d64          = TRUE;
    r.d128         = r.ttl IN {'00xx'};

    TLBI(r);
    if shareability != Shareability_NSH then Broadcast(shareability, r);
    return;
```

Library pseudocode for aarch64/functions/tlbi/AArch64.TLBI_VAA

```
// AArch64.TLBI_VAA()
// ==================
// Invalidate by VA all stage 1 TLB entries in the indicated shareability domain
// matching the indicated VMID (where regime supports VMID) and all ASID in the indicated regime
// with the indicated security state.
// VA and related parameters are derived from Xt.
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.

AArch64.TLBI_VAA(SecurityState security, Regime regime, bits(16) vmid,
                 Shareability shareability, TLBILevel level, TLBIMemAttr attr,  bits(64) Xt)
    assert PSTATE.EL IN {EL3, EL2, EL1};

    TLBIRecord r;
    r.op           = TLBIOp_VAA;
    r.from_aarch64 = TRUE;
    r.security     = security;
    r.regime       = regime;
    r.vmid         = vmid;
    r.level        = level;
    r.attr         = attr;
    r.ttl          = Xt<47:44>;
    r.address      = ZeroExtend(Xt<43:0> : Zeros(12), 64);
    r.d64          = TRUE;
    r.d128         = r.ttl IN {'00xx'};

    TLBI(r);
    if shareability != Shareability_NSH then Broadcast(shareability, r);
    return;
```

Library pseudocode for aarch64/functions/tlbi/AArch64.TLBI_VMALL

```
// AArch64.TLBI_VMALL()
// ====================
// Invalidate all stage 1 entries for the indicated translation regime with the
// the indicated security state for all TLBs within the indicated shareability
// domain that match the indicated VMID (where applicable).
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.
// Note: stage 2 only entries are not in the scope of this operation.

AArch64.TLBI_VMALL(SecurityState security, Regime regime, bits(16) vmid,
                   Shareability shareability, TLBIMemAttr attr)
    assert PSTATE.EL IN {EL3, EL2, EL1};

    TLBIRecord r;
    r.op           = TLBIOp_VMALL;
    r.from_aarch64 = TRUE;
    r.security     = security;
    r.regime       = regime;
    r.level        = TLBILevel_Any;
    r.vmid         = vmid;
    r.attr         = attr;

    TLBI(r);
    if shareability != Shareability_NSH then Broadcast(shareability, r);
    return;
```

Library pseudocode for aarch64/functions/tlbi/AArch64.TLBI_VMALLS12

```
// AArch64.TLBI_VMALLS12()
// =======================
// Invalidate all stage 1 and stage 2 entries for the indicated translation
// regime with the indicated security state for all TLBs within the indicated
// shareability domain that match the indicated VMID.

AArch64.TLBI_VMALLS12(SecurityState security, Regime regime, bits(16) vmid,
                      Shareability shareability, TLBIMemAttr attr)
    assert PSTATE.EL IN {EL3, EL2};

    TLBIRecord r;
    r.op           = TLBIOp_VMALLS12;
    r.from_aarch64 = TRUE;
    r.security     = security;
    r.regime       = regime;
    r.level        = TLBILevel_Any;
    r.vmid         = vmid;
    r.attr         = attr;

    TLBI(r);
    if shareability != Shareability_NSH then Broadcast(shareability, r);
    return;
```

Library pseudocode for aarch64/functions/tlbi/ASID_NONE

```
constant bits(16) ASID_NONE = Zeros(16);
```

Library pseudocode for aarch64/functions/tlbi/Broadcast

```
// Broadcast()
// ===========
// IMPLEMENTATION DEFINED function to broadcast TLBI operation within the indicated shareability
// domain.

Broadcast(Shareability shareability, TLBIRecord r)
    IMPLEMENTATION_DEFINED;
```

Library pseudocode for aarch64/functions/tlbi/DecodeTLBITG

```
// DecodeTLBITG()
// ==============
// Decode translation granule size in TLBI range instructions

TGx DecodeTLBITG(bits(2) tg)
    case tg of
        when '01'   return TGx_4KB;
        when '10'   return TGx_16KB;
        when '11'   return TGx_64KB;
```

Library pseudocode for aarch64/functions/tlbi/GPTTLBIMatch

```
// GPTTLBIMatch()
// ==============
// Determine whether the GPT TLB entry lies within the scope of invalidation

boolean GPTTLBIMatch(TLBIRecord tlbi, GPTEntry entry)
    assert tlbi.op IN {TLBIOp_RPA, TLBIOp_PAALL};

    boolean match;
    bits(64) entry_size_mask    = ZeroExtend(Ones(entry.size), 64);
    bits(64) entry_end_address   = ZeroExtend(entry.pa<55:0> OR entry_size_mask<55:0>, 64);
    bits(64) entry_start_address = ZeroExtend(entry.pa<55:0> AND NOT entry_size_mask<55:0>, 64);

    case tlbi.op of
        when TLBIOp_RPA
            match = (UInt(tlbi.address<55:0>) <= UInt(entry_end_address<55:0>) &&
                     UInt(tlbi.end_address<55:0>) > UInt(entry_start_address<55:0>) &&
                     (tlbi.level == TLBILevel_Any || entry.level == 1));
        when TLBIOp_PAALL
            match = TRUE;

    return match;
```

Library pseudocode for aarch64/functions/tlbi/HasLargeAddress

```
// HasLargeAddress()
// =================
// Returns TRUE if the regime is configured for 52 bit addresses, FALSE otherwise.

boolean HasLargeAddress(Regime regime)
    if !Have52BitIPAAndPASpaceExt() then
        return FALSE;
    case regime of
        when Regime_EL3
            return TCR_EL3<32> == '1';
        when Regime_EL2
            return TCR_EL2<32> == '1';
        when Regime_EL20
            return TCR_EL2<59> == '1';
        when Regime_EL10
            return TCR_EL1<59> == '1';
        otherwise
            Unreachable();
```

Library pseudocode for aarch64/functions/tlbi/ResTLBIRTTL

```
// ResTLBIRTTL()
// =============
// Determine whether the TTL field in TLBI instructions that do apply
// to a range of addresses contains a reserved value

boolean ResTLBIRTTL(bits(2) tg, bits(2) ttl)
    case ttl of
        when '00' return TRUE;
        when '01' return DecodeTLBITG(tg) == TGx_16KB && !Have52BitIPAAndPASpaceExt();
        otherwise return FALSE;
```

Library pseudocode for aarch64/functions/tlbi/ResTLBITTL

```
// ResTLBITTL()
// ============
// Determine whether the TTL field in TLBI instructions that do not apply
// to a range of addresses contains a reserved value

boolean ResTLBITTL(bits(4) ttl)
    case ttl of
        when '00xx' return TRUE;
        when '0100' return !Have52BitIPAAndPASpaceExt();
        when '1000' return TRUE;
        when '1001' return !Have52BitIPAAndPASpaceExt();
        when '1100' return TRUE;
        otherwise   return FALSE;
```

Library pseudocode for aarch64/functions/tlbi/TLBI

```
// TLBI()
// ======
// Invalidates TLB entries for which TLBIMatch() returns TRUE.

TLBI(TLBIRecord r)
    IMPLEMENTATION_DEFINED;
```

Library pseudocode for aarch64/functions/tlbi/TLBILevel

```
// TLBILevel
// =========

enumeration TLBILevel {
    TLBILevel_Any,        // this applies to TLB entries at all levels
    TLBILevel_Last        // this applies to TLB entries at last level only
};
```

Library pseudocode for aarch64/functions/tlbi/TLBIMatch

```
// TLBIMatch()
// ===========
// Determine whether the TLB entry lies within the scope of invalidation

boolean TLBIMatch(TLBIRecord tlbi, TLBRecord entry)
    boolean match;
    bits(64) entry_block_mask    = ZeroExtend(Ones(entry.blocksize), 64);
    bits(64) entry_end_address   = entry.context.ia OR entry_block_mask;
    bits(64) entry_start_address = entry.context.ia AND NOT entry_block_mask;
    case tlbi.op of
        when TLBIOp_DALL, TLBIOp_IALL
            match = (tlbi.security == entry.context.ss &&
                     tlbi.regime   == entry.context.regime);
        when TLBIOp_DASID, TLBIOp_IASID
            match = (entry.context.includes_s1 &&
                     tlbi.security == entry.context.ss &&
                     tlbi.regime   == entry.context.regime &&
                     (!UseVMID(entry.context) || tlbi.vmid == entry.context.vmid) &&
                     (UseASID(entry.context) && entry.context.nG == '1' &&
                        tlbi.asid  == entry.context.asid));
        when TLBIOp_DVA, TLBIOp_IVA
            match = (entry.context.includes_s1 &&
                     tlbi.security == entry.context.ss &&
                     tlbi.regime   == entry.context.regime &&
                     (!UseVMID(entry.context) || tlbi.vmid == entry.context.vmid) &&
                     (!UseASID(entry.context) || tlbi.asid == entry.context.asid ||
                        entry.context.nG == '0') &&
                     tlbi.address<55:entry.blocksize> == entry.context.ia<55:entry.blocksize> &&
                     (tlbi.level == TLBILevel_Any || !entry.walkstate.istable));
        when TLBIOp_ALL
            relax_regime = (tlbi.from_aarch64 &&
                            tlbi.regime IN {Regime_EL20, Regime_EL2} &&
                            entry.context.regime IN {Regime_EL20, Regime_EL2});
            match = (tlbi.security == entry.context.ss &&
                     (tlbi.regime  == entry.context.regime || relax_regime));
        when TLBIOp_ASID
            match = (entry.context.includes_s1 &&
                     tlbi.security == entry.context.ss &&
                     tlbi.regime   == entry.context.regime &&
                     (!UseVMID(entry.context) || tlbi.vmid == entry.context.vmid) &&
                     (UseASID(entry.context) && entry.context.nG == '1' &&
                        tlbi.asid  == entry.context.asid));
        when TLBIOp_IPAS2, TLBIPOp_IPAS2
            match = (!entry.context.includes_s1 && entry.context.includes_s2 &&
                     tlbi.security == entry.context.ss &&
                     tlbi.regime   == entry.context.regime &&
                     (!UseVMID(entry.context) || tlbi.vmid == entry.context.vmid) &&
                     tlbi.ipaspace == entry.context.ipaspace &&
                     tlbi.address<55:entry.blocksize> == entry.context.ia<55:entry.blocksize> &&
                     (!tlbi.from_aarch64 || ResTLBITTL(tlbi.ttl) || (
                         DecodeTLBITG(tlbi.ttl<3:2>) == entry.context.tg &&
                         UInt(tlbi.ttl<1:0>) == entry.walkstate.level)
                     ) &&
                     ((tlbi.d128  && entry.context.isd128) ||
                      (tlbi.d64   && !entry.context.isd128) ||
                      (tlbi.d64 && tlbi.d128)) &&
                     (tlbi.level == TLBILevel_Any || !entry.walkstate.istable));
        when TLBIOp_VAA, TLBIPOp_VAA
            match = (entry.context.includes_s1 &&
                     tlbi.security == entry.context.ss &&
                     tlbi.regime   == entry.context.regime &&
                     (!UseVMID(entry.context) || tlbi.vmid == entry.context.vmid) &&
                     tlbi.address<55:entry.blocksize> == entry.context.ia<55:entry.blocksize> &&
                     (!tlbi.from_aarch64 || ResTLBITTL(tlbi.ttl) || (
                         DecodeTLBITG(tlbi.ttl<3:2>) == entry.context.tg &&
                         UInt(tlbi.ttl<1:0>) == entry.walkstate.level)
                     ) &&
                     ((tlbi.d128  && entry.context.isd128) ||
                      (tlbi.d64   && !entry.context.isd128) ||
                      (tlbi.d64 && tlbi.d128)) &&
```

```
                (tlbi.level == TLBILevel_Any || !entry.walkstate.istable));
        when TLBIOp_VA, TLBIPOp_VA
            match = (entry.context.includes_s1 &&
                     tlbi.security == entry.context.ss &&
                     tlbi.regime   == entry.context.regime &&
                     (!UseVMID(entry.context) || tlbi.vmid == entry.context.vmid) &&
                     (!UseASID(entry.context) || tlbi.asid == entry.context.asid ||
                        entry.context.nG == '0') &&
                     tlbi.address<55:entry.blocksize> == entry.context.ia<55:entry.blocksize> &&
                     (!tlbi.from_aarch64 || ResTLBITTL(tlbi.ttl) || (
                         DecodeTLBITG(tlbi.ttl<3:2>) == entry.context.tg &&
                         UInt(tlbi.ttl<1:0>) == entry.walkstate.level)
                     ) &&
                     ((tlbi.d128  && entry.context.isd128) ||
                      (tlbi.d64  && !entry.context.isd128) ||
                      (tlbi.d64 && tlbi.d128)) &&
                     (tlbi.level == TLBILevel_Any || !entry.walkstate.istable));
        when TLBIOp_VMALL
            match = (entry.context.includes_s1 &&
                     tlbi.security == entry.context.ss &&
                     tlbi.regime   == entry.context.regime &&
                     (!UseVMID(entry.context) || tlbi.vmid == entry.context.vmid));
        when TLBIOp_VMALLS12
            match = (tlbi.security == entry.context.ss &&
                     tlbi.regime   == entry.context.regime &&
                     (!UseVMID(entry.context) || tlbi.vmid == entry.context.vmid));
        when TLBIOp_RIPAS2, TLBIPOp_RIPAS2
            match = (!entry.context.includes_s1 && entry.context.includes_s2 &&
                     tlbi.security == entry.context.ss &&
                     tlbi.regime   == entry.context.regime &&
                     (!UseVMID(entry.context) || tlbi.vmid == entry.context.vmid) &&
                     tlbi.ipaspace == entry.context.ipaspace &&
                     (tlbi.tg != '00' && DecodeTLBITG(tlbi.tg) == entry.context.tg) &&
                     (!tlbi.from_aarch64 || ResTLBIRTTL(tlbi.tg, tlbi.ttl<1:0>) ||
                        UInt(tlbi.ttl<1:0>) == entry.walkstate.level) &&
                     ((tlbi.d128  && entry.context.isd128) ||
                      (tlbi.d64  && !entry.context.isd128) ||
                      (tlbi.d64 && tlbi.d128)) &&
                     UInt(tlbi.address<55:0>) <= UInt(entry_end_address<55:0>) &&
                     UInt(tlbi.end_address<55:0>) > UInt(entry_start_address<55:0>));
        when TLBIOp_RVAA, TLBIPOp_RVAA
            match = (entry.context.includes_s1 &&
                     tlbi.security == entry.context.ss &&
                     tlbi.regime   == entry.context.regime &&
                     (!UseVMID(entry.context) || tlbi.vmid == entry.context.vmid) &&
                     (tlbi.tg != '00' && DecodeTLBITG(tlbi.tg) == entry.context.tg) &&
                     (!tlbi.from_aarch64 || ResTLBIRTTL(tlbi.tg, tlbi.ttl<1:0>) ||
                        UInt(tlbi.ttl<1:0>) == entry.walkstate.level) &&
                     ((tlbi.d128  && entry.context.isd128) ||
                      (tlbi.d64  && !entry.context.isd128) ||
                      (tlbi.d64 && tlbi.d128)) &&
                     UInt(tlbi.address<55:0>) <= UInt(entry_end_address<55:0>) &&
                     UInt(tlbi.end_address<55:0>) > UInt(entry_start_address<55:0>));
        when TLBIOp_RVA, TLBIPOp_RVA
            match = (entry.context.includes_s1 &&
                     tlbi.security == entry.context.ss &&
                     tlbi.regime   == entry.context.regime &&
                     (!UseVMID(entry.context) || tlbi.vmid == entry.context.vmid) &&
                     (!UseASID(entry.context) || tlbi.asid == entry.context.asid ||
                        entry.context.nG == '0') &&
                     (tlbi.tg != '00' && DecodeTLBITG(tlbi.tg) == entry.context.tg) &&
                     (!tlbi.from_aarch64 || ResTLBIRTTL(tlbi.tg, tlbi.ttl<1:0>) ||
                        UInt(tlbi.ttl<1:0>) == entry.walkstate.level) &&
                     ((tlbi.d128  && entry.context.isd128) ||
                      (tlbi.d64  && !entry.context.isd128) ||
                      (tlbi.d64 && tlbi.d128)) &&
                     UInt(tlbi.address<55:0>) <= UInt(entry_end_address<55:0>) &&
                     UInt(tlbi.end_address<55:0>) > UInt(entry_start_address<55:0>));
        when TLBIOp_RPA
            entry_end_address<55:0> = (entry.walkstate.baseaddress.address<55:0> OR
```

```
                                              entry_block_mask<55:0>);
            entry_start_address<55:0> = (entry.walkstate.baseaddress.address<55:0> AND
                                            NOT entry_block_mask<55:0>);
            match = (entry.context.includes_gpt &&
                     UInt(tlbi.address<55:0>) <= UInt(entry_end_address<55:0>) &&
                     UInt(tlbi.end_address<55:0>) > UInt(entry_start_address<55:0>));
        when TLBIOp_PAALL
            match = entry.context.includes_gpt;

    if tlbi.attr == TLBI_ExcludeXS && entry.context.xs == '1' then
        match = FALSE;

    return match;
```

Library pseudocode for aarch64/functions/tlbi/TLBIMemAttr

```
// TLBIMemAttr
// ===========
// Defines the attributes of the memory operations that must be completed in
// order to deem the TLBI operation as completed.

enumeration TLBIMemAttr {
    TLBI_AllAttr,         // All TLB entries within the scope of the invalidation
    TLBI_ExcludeXS        // Only TLB entries with XS=0 within the scope of the invalidation
};
```

Library pseudocode for aarch64/functions/tlbi/TLBIOp

```
// TLBIOp
// ======

enumeration TLBIOp {
    TLBIOp_DALL,          // AArch32 Data TLBI operations - deprecated
    TLBIOp_DASID,
    TLBIOp_DVA,
    TLBIOp_IALL,          // AArch32 Instruction TLBI operations - deprecated
    TLBIOp_IASID,
    TLBIOp_IVA,
    TLBIOp_ALL,
    TLBIOp_ASID,
    TLBIOp_IPAS2,
    TLBIPOp_IPAS2,
    TLBIOp_VAA,
    TLBIOp_VA,
    TLBIPOp_VAA,
    TLBIPOp_VA,
    TLBIOp_VMALL,
    TLBIOp_VMALLS12,
    TLBIOp_RIPAS2,
    TLBIPOp_RIPAS2,
    TLBIOp_RVAA,
    TLBIOp_RVA,
    TLBIPOp_RVAA,
    TLBIPOp_RVA,
    TLBIOp_RPA,
    TLBIOp_PAALL,
};
```

Library pseudocode for aarch64/functions/tlbi/TLBIPRange

```
// TLBIPRange()
// ============
// Extract the input address range information from encoded Xt.

(boolean, bits(2), bits(64), bits(64)) TLBIPRange(Regime regime, bits(128) Xt)
    boolean  valid         = TRUE;
    bits(64) start_address = Zeros(64);
    bits(64) end_address   = Zeros(64);

    bits(2) tg        = Xt<47:46>;
    integer scale     = UInt(Xt<45:44>);
    integer num       = UInt(Xt<43:39>);
    integer tg_bits;

    if tg == '00' then
        return (FALSE, tg, start_address, end_address);

    case tg of
        when '01' // 4KB
            tg_bits = 12;
            start_address<55:12> = Xt<107:64>;
            start_address<63:56> = Replicate(Xt<107>, 8);
        when '10' // 16KB
            tg_bits = 14;
            start_address<55:14> = Xt<107:66>;
            start_address<63:56> = Replicate(Xt<107>, 8);
        when '11' // 64KB
            tg_bits = 16;
            start_address<55:16> = Xt<107:68>;
            start_address<63:56> = Replicate(Xt<107>, 8);
        otherwise
            Unreachable();

    integer range = (num+1) << (5*scale + 1 + tg_bits);
    end_address   = start_address + range<63:0>;

    if end_address<55> != start_address<55> then
        // overflow, saturate it
        end_address = Replicate(start_address<55>, 64-55) : Ones(55);

    return (valid, tg, start_address, end_address);
```

Library pseudocode for aarch64/functions/tlbi/TLBIRange

```
// TLBIRange()
// ===========
// Extract the input address range information from encoded Xt.

(boolean, bits(2), bits(64), bits(64)) TLBIRange(Regime regime, bits(64) Xt)
    boolean  valid = TRUE;
    bits(64) start_address = Zeros(64);
    bits(64) end_address   = Zeros(64);

    bits(2) tg        = Xt<47:46>;
    integer scale     = UInt(Xt<45:44>);
    integer num       = UInt(Xt<43:39>);
    integer tg_bits;

    if tg == '00' then
        return (FALSE, tg, start_address, end_address);

    case tg of
        when '01' // 4KB
            tg_bits = 12;
            if HasLargeAddress(regime) then
                start_address<52:16> = Xt<36:0>;
                start_address<63:53> = Replicate(Xt<36>, 11);
            else
                start_address<48:12> = Xt<36:0>;
                start_address<63:49> = Replicate(Xt<36>, 15);
        when '10' // 16KB
            tg_bits = 14;
            if HasLargeAddress(regime) then
                start_address<52:16> = Xt<36:0>;
                start_address<63:53> = Replicate(Xt<36>, 11);
            else
                start_address<50:14> = Xt<36:0>;
                start_address<63:51> = Replicate(Xt<36>, 13);
        when '11' // 64KB
            tg_bits = 16;
            start_address<52:16> = Xt<36:0>;
            start_address<63:53> = Replicate(Xt<36>, 11);
        otherwise
            Unreachable();

    integer range = (num+1) << (5*scale + 1 + tg_bits);
    end_address   = start_address + range<63:0>;

    if end_address<52> != start_address<52> then
        // overflow, saturate it
        end_address = Replicate(start_address<52>, 64-52) : Ones(52);

    return (valid, tg, start_address, end_address);
```

Library pseudocode for aarch64/functions/tlbi/TLBIRecord

```
// TLBIRecord
// ==========
// Details related to a TLBI operation.

type TLBIRecord is (
    TLBIOp          op,
    boolean         from_aarch64, // originated as an AArch64 operation
    SecurityState   security,
    Regime          regime,
    bits(16)        vmid,
    bits(16)        asid,
    TLBILevel       level,
    TLBIMemAttr     attr,
    PASpace         ipaspace,     // For operations that take IPA as input address
    bits(64)        address,      // input address, for range operations, start address
    bits(64)        end_address,  // for range operations, end address
    boolean         d64,          // For operations that evict VMSAv8-64 based TLB entries
    boolean         d128,         // For operations that evict VMSAv9-128 based TLB entries
    bits(4)         ttl,          // translation table walk level holding the leaf entry
                                  // for the address being invalidated
                                  // For Non-Range Invalidations:
                                  //   When the ttl is
                                  //     '00xx'    : this applies to all TLB entries
                                  //     Otherwise : TLBIP instructions invalidates D128 TLB
                                  //                 entries only
                                  //                 TLBI instructions invalidates D64 TLB
                                  //                 entries only
                                  // For Range Invalidations:
                                  //   When the ttl is
                                  //     '00'      : this applies to all TLB entries
                                  //     Otherwise : TLBIP instructions invalidates D128 TLB
                                  //                 entries only
                                  //                 TLBI instructions invalidates D64 TLB
                                  //                 entries only
    bits(2)         tg            // for range operations, translation granule
)
```

Library pseudocode for aarch64/functions/tlbi/VMID

```
// VMID[]
// ======
// Effective VMID.

bits(16) VMID[]
    if EL2Enabled() then
        if !ELUsingAArch32(EL2) then
            if Have16bitVMID() && VTCR_EL2.VS == '1' then
                return VTTBR_EL2.VMID;
            else
                return ZeroExtend(VTTBR_EL2.VMID<7:0>, 16);
        else
            return ZeroExtend(VTTBR.VMID, 16);
    elsif HaveEL(EL2) && HaveSecureEL2Ext() then
        return Zeros(16);
    else
        return VMID_NONE;
```

Library pseudocode for aarch64/functions/tlbi/VMID_NONE

```
constant bits(16) VMID_NONE = Zeros(16);
```

Library pseudocode for aarch64/functions/tme/CheckTransactionalSystemAccess

```
// CheckTransactionalSystemAccess()
// ================================
// Returns TRUE if an AArch64 MSR, MRS, or SYS instruction is permitted in
// Transactional state, based on the opcode's encoding, and FALSE otherwise.

boolean CheckTransactionalSystemAccess(bits(2) op0, bits(3) op1, bits(4) crn, bits(4) crm,
                                       bits(3) op2, bit read)
    case read:op0:op1:crn:crm:op2 of
        when '0 00 011 0100 xxxx 11x' return TRUE;      // MSR (imm): DAIFSet, DAIFClr
        when '0 01 011 0111 0100 001' return TRUE;      // DC ZVA
        when '0 11 011 0100 0010 00x' return TRUE;      // MSR: NZCV, DAIF
        when '0 11 011 0100 0100 00x' return TRUE;      // MSR: FPCR, FPSR
        when '0 11 000 0100 0110 000' return TRUE;      // MSR: ICC_PMR_EL1
        when '0 11 011 1001 1100 100' return TRUE;      // MRS: PMSWINC_EL0
        when '1 11 011 0010 0101 001'                   // MRS: GCSPR_EL0, at EL0
            return PSTATE.EL == EL0;
        // MRS: GCSPR_EL1 at EL1 OR at EL2 when E2H is '1'
        when '1 11 000 0010 0101 001'
            return PSTATE.EL == EL1 || (PSTATE.EL == EL2 && HCR_EL2.E2H == '1');
        when '1 11 100 0010 0101 001'                   // MRS: GCSPR_EL2, at EL2 when E2H is '0'
            return PSTATE.EL == EL2 && HCR_EL2.E2H == '0';
        when '1 11 110 0010 0101 001'                   // MRS: GCSPR_EL3, at EL3
            return PSTATE.EL == EL3;
        when '0 01 011 0111 0111 000' return TRUE;      // GCSPUSHM
        when '1 01 011 0111 0111 001' return TRUE;      // GCSPOPM
        when '0 01 011 0111 0111 010' return TRUE;      // GCSSS1
        when '1 01 011 0111 0111 011' return TRUE;      // GCSSS2
        when '0 01 000 0111 0111 110' return TRUE;      // GCSPOPX
        when '1 11 101 0010 0101 001' return FALSE;     // MRS: GCSPR_EL12
        when '1 11 000 0010 0101 010' return FALSE;     // MRS: GCSCRE0_EL1
        when '1 11 000 0010 0101 000' return FALSE;     // MRS: GCSCR_EL1
        when '1 11 101 0010 0101 000' return FALSE;     // MRS: GCSCR_EL12
        when '1 11 100 0010 0101 000' return FALSE;     // MRS: GCSCR_EL2
        when '1 11 110 0010 0101 000' return FALSE;     // MRS: GCSCR_EL3
        when '1 11 xxx 0xxx xxxx xxx' return TRUE;      // MRS: op0=3, CRn=0..7
        when '1 11 xxx 100x xxxx xxx' return TRUE;      // MRS: op0=3, CRn=8..9
        when '1 11 xxx 1010 xxxx xxx' return TRUE;      // MRS: op0=3, CRn=10
        when '1 11 000 1100 1x00 010' return TRUE;      // MRS: op0=3, CRn=12 - ICC_HPPIRx_EL1
        when '1 11 000 1100 1011 011' return TRUE;      // MRS: op0=3, CRn=12 - ICC_RPR_EL1
        when '1 11 xxx 1101 xxxx xxx' return TRUE;      // MRS: op0=3, CRn=13
        when '1 11 xxx 1110 xxxx xxx' return TRUE;      // MRS: op0=3, CRn=14
        when '0 01 011 0111 0011 111' return TRUE;      // CPP RCTX
        when '0 01 011 0111 0011 10x' return TRUE;      // CFP RCTX, DVP RCTX
        when 'x 11 xxx 1x11 xxxx xxx'                   // MRS: op1=3, CRn=11,15
            return boolean IMPLEMENTATION_DEFINED;
        otherwise return FALSE;                         // All other SYS, SYSL, MRS, MSR
```

Library pseudocode for aarch64/functions/tme/CommitTransactionalWrites

```
// CommitTransactionalWrites()
// ===========================
// Makes all transactional writes to memory observable by other PEs and reset
// the transactional read and write sets.

CommitTransactionalWrites();
```

Library pseudocode for aarch64/functions/tme/DiscardTransactionalWrites

```
// DiscardTransactionalWrites()
// ============================
// Discards all transactional writes to memory and reset the transactional
// read and write sets.

DiscardTransactionalWrites();
```

Library pseudocode for aarch64/functions/tme/FailTransaction

```
// FailTransaction()
// =================

FailTransaction(TMFailure cause, boolean retry)
    FailTransaction(cause, retry, FALSE, Zeros(15));
    return;

// FailTransaction()
// =================
// Exits Transactional state and discards transactional updates to registers
// and memory.

FailTransaction(TMFailure cause, boolean retry, boolean interrupt, bits(15) reason)
    assert !retry || !interrupt;

    if HaveBRBExt() && BranchRecordAllowed(PSTATE.EL) then BRBFCR_EL1.LASTFAILED = '1';

    DiscardTransactionalWrites();
    // For trivial implementation no transaction checkpoint was taken
    if cause != TMFailure_TRIVIAL then
        RestoreTransactionCheckpoint();
    ClearExclusiveLocal(ProcessorID());

    bits(64) result = Zeros(64);

    result<23> = if interrupt then '1' else '0';
    result<15> = if retry && !interrupt then '1' else '0';
    case cause of
        when TMFailure_TRIVIAL result<24> = '1';
        when TMFailure_DBG     result<22> = '1';
        when TMFailure_NEST    result<21> = '1';
        when TMFailure_SIZE    result<20> = '1';
        when TMFailure_ERR     result<19> = '1';
        when TMFailure_IMP     result<18> = '1';
        when TMFailure_MEM     result<17> = '1';
        when TMFailure_CNCL    result<16> = '1'; result<14:0> = reason;

    TSTATE.depth = 0;
    X[TSTATE.Rt, 64] = result;
    boolean branch_conditional = FALSE;
    BranchTo(TSTATE.nPC, BranchType_TMFAIL, branch_conditional);
    EndOfInstruction();
    return;
```

Library pseudocode for aarch64/functions/tme/IsTMEEnabled

```
// IsTMEEnabled()
// ==============
// Returns TRUE if access to TME instruction is enabled, FALSE otherwise.

boolean IsTMEEnabled()
    if PSTATE.EL IN {EL0, EL1, EL2} && HaveEL(EL3) then
        if SCR_EL3.TME == '0' then
            return FALSE;
    if PSTATE.EL IN {EL0, EL1} && EL2Enabled() then
        if HCR_EL2.TME == '0' then
            return FALSE;
    return TRUE;
```

Library pseudocode for aarch64/functions/tme/MemHasTransactionalAccess

```
// MemHasTransactionalAccess()
// ===========================
// Function checks if transactional accesses are not supported for an address
// range or memory type.

boolean MemHasTransactionalAccess(MemoryAttributes memattrs)
    if ((memattrs.shareability == Shareability_ISH ||
            memattrs.shareability == Shareability_OSH) &&
            memattrs.memtype == MemType_Normal &&
            memattrs.inner.attrs == MemAttr_WB &&
            memattrs.inner.hints == MemHint_RWA &&
            memattrs.inner.transient == FALSE &&
            memattrs.outer.hints == MemHint_RWA &&
            memattrs.outer.attrs == MemAttr_WB &&
            memattrs.outer.transient == FALSE) then
        return TRUE;
    else
        return boolean IMPLEMENTATION_DEFINED "Memory Region does not support Transactional access";
```

Library pseudocode for aarch64/functions/tme/RestoreTransactionCheckpoint

```
// RestoreTransactionCheckpoint()
// ==============================
// Restores part of the PE registers from the transaction checkpoint.

RestoreTransactionCheckpoint()
    SP[]             = TSTATE.SP;
    ICC_PMR_EL1      = TSTATE.ICC_PMR_EL1;
    PSTATE.<N,Z,C,V> = TSTATE.nzcv;
    PSTATE.<D,A,I,F> = TSTATE.<D,A,I,F>;

    for n = 0 to 30
        X[n, 64] = TSTATE.X[n];

    if IsFPEnabled(PSTATE.EL) then
        if IsSVEEnabled(PSTATE.EL) then
            constant integer VL = CurrentVL;
            constant integer PL = VL DIV 8;
            for n = 0 to 31
                Z[n, VL] = TSTATE.Z[n]<VL-1:0>;
            for n = 0 to 15
                P[n, PL] = TSTATE.P[n]<PL-1:0>;
            FFR[PL] = TSTATE.FFR<PL-1:0>;
        else
            for n = 0 to 31
                V[n, 128] = TSTATE.Z[n]<127:0>;
        FPCR = TSTATE.FPCR;
        FPSR = TSTATE.FPSR;

    if HaveGCS() then
        case PSTATE.EL of
            when EL0 GCSPR_EL0 = TSTATE.GCSPR_ELx;
            when EL1 GCSPR_EL1 = TSTATE.GCSPR_ELx;
            when EL2 GCSPR_EL2 = TSTATE.GCSPR_ELx;
            when EL3 GCSPR_EL3 = TSTATE.GCSPR_ELx;

    return;
```

Library pseudocode for aarch64/functions/tme/StartTrackingTransactionalReadsWrites

```
// StartTrackingTransactionalReadsWrites()
// =======================================
// Starts tracking transactional reads and writes to memory.

StartTrackingTransactionalReadsWrites();
```

Library pseudocode for aarch64/functions/tme/TMFailure

```
// TMFailure
// =========
// Transactional failure causes

enumeration TMFailure {
    TMFailure_CNCL,    // Executed a TCANCEL instruction
    TMFailure_DBG,     // A debug event was generated
    TMFailure_ERR,     // A non-permissible operation was attempted
    TMFailure_NEST,    // The maximum transactional nesting level was exceeded
    TMFailure_SIZE,    // The transactional read or write set limit was exceeded
    TMFailure_MEM,     // A transactional conflict occurred
    TMFailure_TRIVIAL, // Only a TRIVIAL version of TM is available
    TMFailure_IMP      // Any other failure cause
};
```

Library pseudocode for aarch64/functions/tme/TMState

```
// TMState
// =======
// Transactional execution state bits.
// There is no significance to the field order.

type TMState is (
    integer       depth,              // Transaction nesting depth
    integer       Rt,                 // TSTART destination register
    bits(64)      nPC,                // Fallback instruction address
    array[0..30] of bits(64)     X,   // General purpose registers
    array[0..31] of bits(MAX_VL) Z,   // Vector registers
    array[0..15] of bits(MAX_PL) P,   // Predicate registers
    bits(MAX_PL)  FFR,                // First Fault Register
    bits(64)      SP,                 // Stack Pointer at current EL
    bits(64)      FPCR,               // Floating-point Control Register
    bits(64)      FPSR,               // Floating-point Status Register
    bits(64)      ICC_PMR_EL1,        // Interrupt Controller Interrupt Priority Mask Register
    bits(64)      GCSPR_ELx,          // GCS pointer for current EL
    bits(4)       nzcv,               // Condition flags
    bits(1)       D,                  // Debug mask bit
    bits(1)       A,                  // SError interrupt mask bit
    bits(1)       I,                  // IRQ mask bit
    bits(1)       F                   // FIQ mask bit
)
```

Library pseudocode for aarch64/functions/tme/TSTATE

```
TMState TSTATE;
```

Library pseudocode for aarch64/functions/tme/TakeTransactionCheckpoint

```
// TakeTransactionCheckpoint()
// ===========================
// Captures part of the PE registers into the transaction checkpoint.

TakeTransactionCheckpoint()
    TSTATE.SP              = SP[];
    TSTATE.ICC_PMR_EL1     = ICC_PMR_EL1;
    TSTATE.nzcv            = PSTATE.<N,Z,C,V>;
    TSTATE.<D,A,I,F>       = PSTATE.<D,A,I,F>;

    for n = 0 to 30
        TSTATE.X[n] = X[n, 64];

    if IsFPEnabled(PSTATE.EL) then
        if IsSVEEnabled(PSTATE.EL) then
            constant integer VL = CurrentVL;
            constant integer PL = VL DIV 8;
            for n = 0 to 31
                TSTATE.Z[n]<VL-1:0> = Z[n, VL];
            for n = 0 to 15
                TSTATE.P[n]<PL-1:0> = P[n, PL];
            TSTATE.FFR<PL-1:0> = FFR[PL];
        else
            for n = 0 to 31
                TSTATE.Z[n]<127:0> = V[n, 128];
        TSTATE.FPCR = FPCR;
        TSTATE.FPSR = FPSR;

    if HaveGCS() then
        case PSTATE.EL of
            when EL0 TSTATE.GCSPR_ELx = GCSPR_EL0;
            when EL1 TSTATE.GCSPR_ELx = GCSPR_EL1;
            when EL2 TSTATE.GCSPR_ELx = GCSPR_EL2;
            when EL3 TSTATE.GCSPR_ELx = GCSPR_EL3;

    return;
```

Library pseudocode for aarch64/functions/tme/TransactionStartTrap

```
// TransactionStartTrap()
// ======================
// Traps the execution of TSTART instruction.

TransactionStartTrap(integer dreg)
    bits(2) targetEL;
    bits(64) preferred_exception_return = ThisInstrAddr(64);
    vect_offset = 0x0;

    except = ExceptionSyndrome(Exception_TSTARTAccessTrap);
    except.syndrome<9:5> = dreg<4:0>;

    if UInt(PSTATE.EL) > UInt(EL1) then
        targetEL = PSTATE.EL;
    elsif EL2Enabled() && HCR_EL2.TGE == '1' then
        targetEL = EL2;
    else
        targetEL = EL1;
    AArch64.TakeException(targetEL, except, preferred_exception_return, vect_offset);
```

Library pseudocode for aarch64/functions/vbitop/VBitOp

```
// VBitOp
// ======
// Vector bit select instruction types.

enumeration VBitOp      {VBitOp_VBIF, VBitOp_VBIT, VBitOp_VBSL, VBitOp_VEOR};
```

Library pseudocode for aarch64/translation/attrs/AArch64.MAIRAttr

```
// AArch64.MAIRAttr()
// ==================
// Retrieve the memory attribute encoding indexed in the given MAIR

bits(8) AArch64.MAIRAttr(integer index, MAIRType mair2, MAIRType mair)
    bit_index = 8 * index;
    assert (index < 8 || (HaveAIEExt() && (index < 16)));
    if (index > 7) then
        bit_index = bit_index - 64;                     // Read from LSB at MAIR2
        return mair2<bit_index+7:bit_index>;
    else
        return mair<bit_index+7:bit_index>;
```

Library pseudocode for aarch64/translation/debug/AArch64.CheckBreakpoint

```
// AArch64.CheckBreakpoint()
// =========================
// Called before executing the instruction of length "size" bytes at "vaddress" in an AArch64
// translation regime, when either debug exceptions are enabled, or halting debug is enabled
// and halting is allowed.

FaultRecord AArch64.CheckBreakpoint(FaultRecord fault_in, bits(64) vaddress,
                                    AccessDescriptor accdesc, integer size)
    assert !ELUsingAArch32(S1TranslationRegime());
    assert (UsingAArch32() && size IN {2,4}) || size == 4;

    FaultRecord fault = fault_in;
    boolean match = FALSE;
    boolean mismatch = TRUE;            // Default assumption that all mismatches are outside
                                        // the range of all address match breakpoints
    boolean mismatch_bp = FALSE;        // Has a breakpoint been configured for a mismatch

    for i = 0 to NumBreakpointsImplemented() - 1
        (match_i, is_mismatch_i) = AArch64.BreakpointMatch(i, vaddress, accdesc, size);
        if is_mismatch_i then
            mismatch_bp = TRUE;
            mismatch    = mismatch && !match_i;
        else
            match = match || match_i;

    if match || (mismatch && mismatch_bp) then
        fault.statuscode = Fault_Debug;
        if HaltOnBreakpointOrWatchpoint() then
            reason = DebugHalt_Breakpoint;
            Halt(reason);

    return fault;
```

Library pseudocode for aarch64/translation/debug/AArch64.CheckDebug

```
// AArch64.CheckDebug()
// ====================
// Called on each access to check for a debug exception or entry to Debug state.

FaultRecord AArch64.CheckDebug(bits(64) vaddress, AccessDescriptor accdesc, integer size)

    FaultRecord fault = NoFault(accdesc);
    boolean generate_exception;

    boolean d_side = (IsDataAccess(accdesc.acctype) || accdesc.acctype == AccessType_DC);
    boolean i_side = (accdesc.acctype == AccessType_IFETCH);
    if accdesc.acctype == AccessType_NV2 then
        mask = '0';
        ss = CurrentSecurityState();
        generate_exception = (AArch64.GenerateDebugExceptionsFrom(EL2, ss, mask) &&
                              MDSCR_EL1.MDE == '1');
    else
        generate_exception = AArch64.GenerateDebugExceptions() && MDSCR_EL1.MDE == '1';
    halt = HaltOnBreakpointOrWatchpoint();

    if generate_exception || halt then
        if d_side then
            fault = AArch64.CheckWatchpoint(fault, vaddress, accdesc, size);
        elsif i_side then
            fault = AArch64.CheckBreakpoint(fault, vaddress, accdesc, size);

    return fault;
```

Library pseudocode for aarch64/translation/debug/AArch64.CheckWatchpoint

```
// AArch64.CheckWatchpoint()
// =========================
// Called before accessing the memory location of "size" bytes at "address",
// when either debug exceptions are enabled for the access, or halting debug
// is enabled and halting is allowed.

FaultRecord AArch64.CheckWatchpoint(FaultRecord fault_in, bits(64) vaddress,
                                    AccessDescriptor accdesc, integer size)
    assert !ELUsingAArch32(S1TranslationRegime());
    FaultRecord fault = fault_in;

    if accdesc.acctype == AccessType_DC then
        if accdesc.cacheop != CacheOp_Invalidate then
            return fault;
    elsif !IsDataAccess(accdesc.acctype) then
        return fault;

    for i = 0 to NumWatchpointsImplemented() - 1
        if AArch64.WatchpointMatch(i, vaddress, size, accdesc) then
            fault.statuscode = Fault_Debug;
            if DBGWCR_EL1[i].LSC<0> == '1' && accdesc.read then
                fault.write = FALSE;
            elsif DBGWCR_EL1[i].LSC<1> == '1' && accdesc.write then
                fault.write = TRUE;

    if (fault.statuscode == Fault_Debug && HaltOnBreakpointOrWatchpoint() &&
            !accdesc.nonfault && !(accdesc.firstfault && !accdesc.first)) then
        reason = DebugHalt_Watchpoint;
        EDWAR = vaddress;
        Halt(reason);

    return fault;
```

Library pseudocode for aarch64/translation/vmsa_addrcalc/AArch64.IASize

```
// AArch64.IASize()
// ================
// Retrieve the number of bits containing the input address

integer AArch64.IASize(bits(6) txsz)
    return 64 - UInt(txsz);
```

Library pseudocode for aarch64/translation/vmsa_addrcalc/AArch64.LeafBase

```
// AArch64.LeafBase()
// ==================
// Extract the address embedded in a block and page descriptor pointing to the
// base of a memory block

bits(56) AArch64.LeafBase(bits(N) descriptor, bit d128, bit ds,
                              TGx tgx, integer level)
    bits(56) leafbase = Zeros(56);

    granulebits  = TGxGranuleBits(tgx);
    descsizelog2 = if d128 == '1' then 4 else 3;
    stride      = granulebits - descsizelog2;
    leafsize    = granulebits + stride * (FINAL_LEVEL - level);

    leafbase<47:0> = descriptor<47:leafsize>:Zeros(leafsize);

    if Have56BitPAExt() && d128 == '1' then
        leafbase<55:48> = descriptor<55:48>;
        return leafbase;
    if Have52BitPAExt() && tgx == TGx_64KB then
        leafbase<51:48> = descriptor<15:12>;
    elsif ds == '1' then
        leafbase<51:48> = descriptor<9:8>:descriptor<49:48>;

    return leafbase;
```

Library pseudocode for aarch64/translation/vmsa_addrcalc/AArch64.NextTableBase

```
// AArch64.NextTableBase()
// =======================
// Extract the address embedded in a table descriptor pointing to the base of
// the next level table of descriptors

bits(56) AArch64.NextTableBase(bits(N) descriptor, bit d128, bit ds, TGx tgx)
    bits(56) tablebase = Zeros(56);

    case tgx of
        when TGx_4KB  tablebase<47:12> = descriptor<47:12>;
        when TGx_16KB tablebase<47:14> = descriptor<47:14>;
        when TGx_64KB tablebase<47:16> = descriptor<47:16>;

    if Have56BitPAExt() && d128 == '1' then
        tablebase<55:48> = descriptor<55:48>;
        return tablebase;
    if Have52BitPAExt() && tgx == TGx_64KB then
        tablebase<51:48> = descriptor<15:12>;
        return tablebase;
    if ds == '1' then
        tablebase<51:48> = descriptor<9:8>:descriptor<49:48>;
        return tablebase;
    return tablebase;
```

Library pseudocode for aarch64/translation/vmsa_addrcalc/AArch64.PhysicalAddressSize

```
// AArch64.PhysicalAddressSize()
// =============================
// Retrieve the number of bits bounding the physical address

integer AArch64.PhysicalAddressSize(bit d128, bits(3) encoded_ps, TGx tgx)
    integer ps;
    integer max_ps;

    case encoded_ps of
        when '000'  ps = 32;
        when '001'  ps = 36;
        when '010'  ps = 40;
        when '011'  ps = 42;
        when '100'  ps = 44;
        when '101'  ps = 48;
        when '110'  ps = 52;
        when '111'  ps = 56;
    if !Have56BitPAExt() || d128 == '0' then
        if tgx != TGx_64KB && !Have52BitIPAAndPASpaceExt() then
            max_ps = Min(48, AArch64.PAMax());
        elsif !Have52BitPAExt() then
            max_ps = Min(48, AArch64.PAMax());
        else
            max_ps = Min(52, AArch64.PAMax());
    else
        max_ps = AArch64.PAMax();

    return Min(ps, max_ps);
```

Library pseudocode for aarch64/translation/vmsa_addrcalc/AArch64.S1SLTTEntryAddress

```
// AArch64.S1SLTTEntryAddress()
// ============================
// Compute the first stage 1 translation table descriptor address within the
// table pointed to by the base at the start level

FullAddress AArch64.S1SLTTEntryAddress(integer level, S1TTWParams walkparams,
                                       bits(64) ia, FullAddress tablebase)
    // Input Address size
    iasize       = AArch64.IASize(walkparams.txsz);
    granulebits  = TGxGranuleBits(walkparams.tgx);
    descsizelog2 = if walkparams.d128 == '1' then 4 else 3;
    stride       = granulebits - descsizelog2;
    levels       = FINAL_LEVEL - level;

    bits(56) index;
    lsb   = levels*stride + granulebits;
    msb   = iasize - 1;
    index = ZeroExtend(ia<msb:lsb>:Zeros(descsizelog2), 56);

    FullAddress descaddress;
    descaddress.address = tablebase.address OR index;
    descaddress.paspace = tablebase.paspace;

    return descaddress;
```

Library pseudocode for aarch64/translation/vmsa_addrcalc/AArch64.S1StartLevel

```
// AArch64.S1StartLevel()
// ======================
// Compute the initial lookup level when performing a stage 1 translation
// table walk

integer AArch64.S1StartLevel(S1TTWParams walkparams)
    // Input Address size
    iasize       = AArch64.IASize(walkparams.txsz);
    granulebits  = TGxGranuleBits(walkparams.tgx);
    descsizelog2 = if walkparams.d128 == '1' then 4 else 3;
    stride       = granulebits - descsizelog2;
    s1startlevel = FINAL_LEVEL - (((iasize-1) - granulebits) DIV stride);
    if walkparams.d128 == '1' then
        s1startlevel = s1startlevel + UInt(walkparams.skl);
    return s1startlevel;
```

Library pseudocode for aarch64/translation/vmsa_addrcalc/AArch64.S1TTBaseAddress

```
// AArch64.S1TTBaseAddress()
// =========================
// Retrieve the PA/IPA pointing to the base of the initial translation table of stage 1

bits(56) AArch64.S1TTBaseAddress(S1TTWParams walkparams, Regime regime, bits(N) ttbr)
    bits(56) tablebase = Zeros(56);

    // Input Address size
    iasize      = AArch64.IASize(walkparams.txsz);
    granulebits = TGxGranuleBits(walkparams.tgx);
    descsizelog2 = if walkparams.d128 == '1' then 4 else 3;
    stride      = granulebits - descsizelog2;
    startlevel  = AArch64.S1StartLevel(walkparams);
    levels      = FINAL_LEVEL - startlevel;

    // Base address is aligned to size of the initial translation table in bytes
    tsize = (iasize - (levels*stride + granulebits)) + descsizelog2;

    if Have56BitPAExt() &&  walkparams.d128 == '1' then
        tsize = Max(tsize, 5);
        if regime == Regime_EL3 then
            tablebase<55:5> = ttbr<55:5>;
        else
            tablebase<55:5> = ttbr<87:80>:ttbr<47:5>;
    elsif ((Have52BitPAExt() && walkparams.tgx == TGx_64KB && walkparams.ps == '110') ||
            (walkparams.ds == '1')) then
        tsize = Max(tsize, 6);
        tablebase<51:6> = ttbr<5:2>:ttbr<47:6>;
    else
        tablebase<47:1> = ttbr<47:1>;
    tablebase = Align(tablebase, 1 << tsize);
    return tablebase;
```

Library pseudocode for aarch64/translation/vmsa_addrcalc/AArch64.S2SLTTEntryAddress

```
// AArch64.S2SLTTEntryAddress()
// ============================
// Compute the first stage 2 translation table descriptor address within the
// table pointed to by the base at the start level

FullAddress AArch64.S2SLTTEntryAddress(S2TTWParams walkparams, bits(56) ipa,
                                       FullAddress tablebase)
    startlevel   = AArch64.S2StartLevel(walkparams);
    iasize       = AArch64.IASize(walkparams.txsz);
    granulebits  = TGxGranuleBits(walkparams.tgx);
    descsizelog2 = if walkparams.d128 == '1' then 4 else 3;
    stride       = granulebits - descsizelog2;
    levels       = FINAL_LEVEL - startlevel;

    bits(56) index;
    integer lsb;
    integer msb;
    lsb   = levels*stride + granulebits;
    msb   = iasize - 1;
    index = ZeroExtend(ipa<msb:lsb>:Zeros(descsizelog2), 56);

    FullAddress descaddress;
    descaddress.address = tablebase.address OR index;
    descaddress.paspace = tablebase.paspace;

    return descaddress;
```

Library pseudocode for aarch64/translation/vmsa_addrcalc/AArch64.S2StartLevel

```
// AArch64.S2StartLevel()
// ======================
// Determine the initial lookup level when performing a stage 2 translation
// table walk

integer AArch64.S2StartLevel(S2TTWParams walkparams)
    if walkparams.d128 == '1' then
        iasize       = AArch64.IASize(walkparams.txsz);
        granulebits  = TGxGranuleBits(walkparams.tgx);
        descsizelog2 = 4;
        stride       = granulebits - descsizelog2;
        s2startlevel = FINAL_LEVEL - (((iasize-1) - granulebits) DIV stride);
        s2startlevel = s2startlevel + UInt(walkparams.skl);

        return s2startlevel;

    case walkparams.tgx of
        when TGx_4KB
            case walkparams.sl2:walkparams.sl0 of
                when '000' return 2;
                when '001' return 1;
                when '010' return 0;
                when '011' return 3;
                when '100' return -1;
        when TGx_16KB
            case walkparams.sl0 of
                when '00' return 3;
                when '01' return 2;
                when '10' return 1;
                when '11' return 0;
        when TGx_64KB
            case walkparams.sl0 of
                when '00' return 3;
                when '01' return 2;
                when '10' return 1;
```

Library pseudocode for aarch64/translation/vmsa_addrcalc/AArch64.S2TTBaseAddress

```
// AArch64.S2TTBaseAddress()
// =========================
// Retrieve the PA/IPA pointing to the base of the initial translation table of stage 2

bits(56) AArch64.S2TTBaseAddress(S2TTWParams walkparams, PASpace paspace, bits(N) ttbr)
    bits(56) tablebase = Zeros(56);

    // Input Address size
    iasize      = AArch64.IASize(walkparams.txsz);
    granulebits = TGxGranuleBits(walkparams.tgx);
    descsizelog2 = if walkparams.d128 == '1' then 4 else 3;
    stride      = granulebits - descsizelog2;
    startlevel  = AArch64.S2StartLevel(walkparams);
    levels      = FINAL_LEVEL - startlevel;

    // Base address is aligned to size of the initial translation table in bytes
    tsize = (iasize - (levels*stride + granulebits)) + descsizelog2;

    if Have56BitPAExt() &&  walkparams.d128 == '1' then
        tsize = Max(tsize, 5);
        if paspace == PAS_Secure then
            tablebase<55:5> = ttbr<55:5>;
        else
            tablebase<55:5> = ttbr<87:80>:ttbr<47:5>;
    elsif ((Have52BitPAExt() && walkparams.tgx == TGx_64KB && walkparams.ps == '110') ||
            (walkparams.ds == '1')) then
        tsize = Max(tsize, 6);
        tablebase<51:6> = ttbr<5:2>:ttbr<47:6>;
    else
        tablebase<47:1> = ttbr<47:1>;
    tablebase = Align(tablebase, 1 << tsize);
    return tablebase;
```

Library pseudocode for aarch64/translation/vmsa_addrcalc/AArch64.TTEntryAddress

```
// AArch64.TTEntryAddress()
// ========================
// Compute translation table descriptor address within the table pointed to by
// the table base

FullAddress AArch64.TTEntryAddress(integer level, bit d128, bits(2) skl, TGx tgx, bits(6) txsz,
                                   bits(64) ia, FullAddress tablebase)
    // Input Address size
    iasize      = AArch64.IASize(txsz);
    granulebits = TGxGranuleBits(tgx);
    descsizelog2 = if d128 == '1' then 4 else 3;
    stride      = granulebits - descsizelog2;
    levels      = FINAL_LEVEL - level;

    bits(56) index;
    integer lsb;
    integer msb;

    lsb   = levels*stride + granulebits;
    if d128 == '1' then
        msb   = (lsb + stride*(1 + UInt(skl))) - 1;
    else
        msb   = (lsb + stride) - 1;
    index = ZeroExtend(ia<msb:lsb>:Zeros(descsizelog2), 56);

    FullAddress descaddress;
    descaddress.address = tablebase.address OR index;
    descaddress.paspace = tablebase.paspace;

    return descaddress;
```

Library pseudocode for aarch64/translation/vmsa_faults/AArch64.AddrTop

```
// AArch64.AddrTop()
// =================
// Get the top bit position of the virtual address.
// Bits above are not accounted as part of the translation process.

integer AArch64.AddrTop(bit tbid, AccessType acctype, bit tbi)
    if tbid == '1' && acctype == AccessType_IFETCH then
        return 63;

    if tbi == '1' then
        return 55;
    else
        return 63;
```

Library pseudocode for aarch64/translation/vmsa_faults/AArch64.ContiguousBitFaults

```
// AArch64.ContiguousBitFaults()
// =============================
// If contiguous bit is set, returns whether the translation size exceeds the
// input address size and if the implementation generates a fault

boolean AArch64.ContiguousBitFaults(bit d128, bits(6) txsz, TGx tgx, integer level)
    // Input Address size
    iasize = AArch64.IASize(txsz);
    // Translation size
    tsize  = TranslationSize(d128, tgx, level) + ContiguousSize(d128, tgx, level);

    return (tsize > iasize &&
            boolean IMPLEMENTATION_DEFINED "Translation fault on misprogrammed contiguous bit");
```

Library pseudocode for aarch64/translation/vmsa_faults/AArch64.IPAIsOutOfRange

```
// AArch64.IPAIsOutOfRange()
// =========================
// Check bits not resolved by translation are ZERO

boolean AArch64.IPAIsOutOfRange(bits(56) ipa, S2TTWParams walkparams)
    //Input Address size
    iasize = AArch64.IASize(walkparams.txsz);

    if iasize < 56 then
        return !IsZero(ipa<55:iasize>);
    else
        return FALSE;
```

Library pseudocode for aarch64/translation/vmsa_faults/AArch64.OAOutOfRange

```
// AArch64.OAOutOfRange()
// ======================
// Returns whether output address is expressed in the configured size number of bits

boolean AArch64.OAOutOfRange(bits(56) address, bit d128, bits(3) ps, TGx tgx)
    // Output Address size
    oasize  = AArch64.PhysicalAddressSize(d128, ps, tgx);

    if oasize < 56 then
        return !IsZero(address<55:oasize>);
    else
        return FALSE;
```

Library pseudocode for aarch64/translation/vmsa_faults/AArch64.S1CheckPermissions

```
// AArch64.S1CheckPermissions()
// ============================
// Checks whether stage 1 access violates permissions of target memory
// and returns a fault record

FaultRecord AArch64.S1CheckPermissions(FaultRecord fault_in, Regime regime, TTWState walkstate,
                                       S1TTWParams walkparams, AccessDescriptor accdesc)
    FaultRecord fault = fault_in;
    Permissions permissions = walkstate.permissions;
    S1AccessControls s1perms;

    s1perms = AArch64.S1ComputePermissions(regime, walkstate, walkparams, accdesc);

    if accdesc.acctype == AccessType_IFETCH then
        if s1perms.overlay && s1perms.ox == '0' then
            fault.statuscode = Fault_Permission;
            fault.overlay    = TRUE;
        elsif (walkstate.memattrs.memtype == MemType_Device &&
                ConstrainUnpredictable(Unpredictable_INSTRDEVICE) == Constraint_FAULT) then
            fault.statuscode = Fault_Permission;
        elsif s1perms.x == '0' then
            fault.statuscode = Fault_Permission;
    elsif accdesc.acctype == AccessType_DC then
        if accdesc.cacheop == CacheOp_Invalidate then
            if s1perms.overlay && s1perms.ow == '0' then
                fault.statuscode = Fault_Permission;
                fault.overlay    = TRUE;
            elsif s1perms.w == '0' then
                fault.statuscode = Fault_Permission;
        // DC from privileged context which clean cannot generate a Permission fault
        elsif accdesc.el == EL0 then
            if s1perms.overlay && s1perms.or == '0' then
                fault.statuscode = Fault_Permission;
                fault.overlay    = TRUE;
            elsif (walkparams.cmow == '1' &&
                    accdesc.opscope == CacheOpScope_PoC &&
                    accdesc.cacheop == CacheOp_CleanInvalidate &&
                    s1perms.overlay && s1perms.ow == '0') then
                fault.statuscode = Fault_Permission;
                fault.overlay    = TRUE;
            elsif s1perms.r == '0' then
                fault.statuscode = Fault_Permission;
            elsif (walkparams.cmow == '1' &&
                    accdesc.opscope == CacheOpScope_PoC &&
                    accdesc.cacheop == CacheOp_CleanInvalidate &&
                    s1perms.w == '0') then
                fault.statuscode = Fault_Permission;
    elsif accdesc.acctype == AccessType_IC then
        // IC from privileged context cannot generate Permission fault
        if accdesc.el == EL0 then
            if (s1perms.overlay && s1perms.or == '0' &&
                  boolean IMPLEMENTATION_DEFINED "Permission fault on EL0 IC_IVAU execution") then
                fault.statuscode = Fault_Permission;
                fault.overlay    = TRUE;
            elsif walkparams.cmow == '1' && s1perms.overlay && s1perms.ow == '0' then
                fault.statuscode = Fault_Permission;
                fault.overlay    = TRUE;
            elsif (s1perms.r == '0' &&
                  boolean IMPLEMENTATION_DEFINED "Permission fault on EL0 IC_IVAU execution") then
                fault.statuscode = Fault_Permission;
            elsif walkparams.cmow == '1' && s1perms.w == '0' then
                fault.statuscode = Fault_Permission;
    elsif HaveGCS() && accdesc.acctype == AccessType_GCS then
        if s1perms.gcs == '0' then
            fault.statuscode = Fault_Permission;
        elsif accdesc.write && walkparams.<ha,hd> != '11' && permissions.ndirty == '1' then
            fault.statuscode = Fault_Permission;
            fault.dirtybit   = TRUE;
            fault.write      = TRUE;
    elsif accdesc.read && s1perms.overlay && s1perms.or == '0' then
```

```
        fault.statuscode = Fault_Permission;
        fault.overlay    = TRUE;
        fault.write      = FALSE;
    elsif accdesc.write && s1perms.overlay && s1perms.ow == '0' then
        fault.statuscode = Fault_Permission;
        fault.overlay    = TRUE;
        fault.write      = TRUE;
    elsif accdesc.read && s1perms.r == '0' then
        fault.statuscode = Fault_Permission;
        fault.write      = FALSE;
    elsif accdesc.write && s1perms.w == '0' then
        fault.statuscode = Fault_Permission;
        fault.write      = TRUE;
    elsif (accdesc.write && accdesc.tagaccess &&
            walkstate.memattrs.tags == MemTag_CanonicallyTagged) then
        fault.statuscode   = Fault_Permission;
        fault.write        = TRUE;
        fault.s1tagnotdata = TRUE;
    elsif (accdesc.write && !(walkparams.<ha,hd> == '11') && walkparams.pie == '1' &&
            permissions.ndirty == '1') then
        fault.statuscode = Fault_Permission;
        fault.dirtybit   = TRUE;
        fault.write      = TRUE;

    return fault;
```

Library pseudocode for aarch64/translation/vmsa_faults/AArch64.S1ComputePermissions

```
// AArch64.S1ComputePermissions()
// ==============================
// Computes the overall stage 1 permissions

S1AccessControls AArch64.S1ComputePermissions(Regime regime, TTWState walkstate,
                                              S1TTWParams walkparams, AccessDescriptor accdesc)
    Permissions permissions = walkstate.permissions;
    S1AccessControls s1perms;

    if walkparams.pie == '1' then
        s1perms = AArch64.S1IndirectBasePermissions(regime, walkstate, walkparams, accdesc);
    else
        s1perms = AArch64.S1DirectBasePermissions(regime, walkstate, walkparams, accdesc);

    if accdesc.el == EL0 && !AArch64.S1E0POEnabled(regime, walkparams.nv1) then
        s1perms.overlay = FALSE;
    elsif accdesc.el != EL0 && !AArch64.S1POEnabled(regime) then
        s1perms.overlay = FALSE;

    if s1perms.overlay then
        s1overlay_perms = AArch64.S1OverlayPermissions(regime, walkstate, accdesc);
        s1perms.or = s1overlay_perms.or;
        s1perms.ow = s1overlay_perms.ow;
        s1perms.ox = s1overlay_perms.ox;

    // If wxn is set, overlay execute permissions is set to 0
    if s1perms.overlay && s1perms.wxn == '1' && s1perms.ox == '1' then
        s1perms.ow = '0';
    elsif s1perms.wxn == '1' then
        s1perms.x = '0';

    return s1perms;
```

Library pseudocode for aarch64/translation/vmsa_faults/AArch64.S1DirectBasePermissions

```
// AArch64.S1DirectBasePermissions()
// =================================
// Computes the stage 1 direct base permissions

S1AccessControls AArch64.S1DirectBasePermissions(Regime regime, TTWState walkstate,
                                                 S1TTWParams walkparams, AccessDescriptor accdesc)
    bit  r,  w,  x;
    bit pr, pw, px;
    bit ur, uw, ux;
    Permissions permissions = walkstate.permissions;
    S1AccessControls s1perms;

    if HasUnprivileged(regime) then
        // Apply leaf permissions
        case permissions.ap<2:1> of
            when '00' (pr,pw,ur,uw) = ('1','1','0','0'); // Privileged access
            when '01' (pr,pw,ur,uw) = ('1','1','1','1'); // No effect
            when '10' (pr,pw,ur,uw) = ('1','0','0','0'); // Read-only, privileged access
            when '11' (pr,pw,ur,uw) = ('1','0','1','0'); // Read-only

        // Apply hierarchical permissions
        case permissions.ap_table of
            when '00' (pr,pw,ur,uw) = ( pr, pw, ur, uw); // No effect
            when '01' (pr,pw,ur,uw) = ( pr, pw,'0','0'); // Privileged access
            when '10' (pr,pw,ur,uw) = ( pr,'0', ur,'0'); // Read-only
            when '11' (pr,pw,ur,uw) = ( pr,'0','0','0'); // Read-only, privileged access

        // Locations writable by unprivileged cannot be executed by privileged
        px = NOT(permissions.pxn OR permissions.pxn_table OR uw);
        ux = NOT(permissions.uxn OR permissions.uxn_table);

        if HavePANExt() && accdesc.pan && !(regime == Regime_EL10 && walkparams.nv1 == '1') then
            bit pan;
            if (boolean IMPLEMENTATION_DEFINED "SCR_EL3.SIF affects EPAN" &&
                    accdesc.ss == SS_Secure &&
                    walkstate.baseaddress.paspace == PAS_NonSecure &&
                    walkparams.sif == '1') then
                ux = '0';

            if (boolean IMPLEMENTATION_DEFINED "Realm EL2&0 regime affects EPAN" &&
                    accdesc.ss == SS_Realm && regime == Regime_EL20 &&
                    walkstate.baseaddress.paspace != PAS_Realm) then
                ux = '0';

            pan = PSTATE.PAN AND (ur OR uw OR (walkparams.epan AND ux));
            pr = pr AND NOT(pan);
            pw = pw AND NOT(pan);

    else
        // Apply leaf permissions
        case permissions.ap<2> of
            when '0' (pr,pw) = ('1','1'); // No effect
            when '1' (pr,pw) = ('1','0'); // Read-only

        // Apply hierarchical permissions
        case permissions.ap_table<1> of
            when '0' (pr,pw) = ( pr, pw); // No effect
            when '1' (pr,pw) = ( pr,'0'); // Read-only

        px = NOT(permissions.xn OR permissions.xn_table);

    (r,w,x) = if accdesc.el == EL0 then (ur,uw,ux) else (pr,pw,px);

    // Compute WXN value
    wxn = walkparams.wxn AND w AND x;

    // Prevent execution from Non-secure space by PE in secure state if SIF is set
    if accdesc.ss == SS_Secure && walkstate.baseaddress.paspace == PAS_NonSecure then
        x = x AND NOT(walkparams.sif);
    // Prevent execution from non-Root space by Root
```

```
    if accdesc.ss == SS_Root && walkstate.baseaddress.paspace != PAS_Root then
        x = '0';
    // Prevent execution from non-Realm space by Realm EL2 and Realm EL2&0
    if (accdesc.ss == SS_Realm && regime IN {Regime_EL2, Regime_EL20} &&
            walkstate.baseaddress.paspace != PAS_Realm) then
        x = '0';

    s1perms.r   = r;
    s1perms.w   = w;
    s1perms.x   = x;
    s1perms.gcs = '0';
    s1perms.wxn = wxn;
    s1perms.overlay = TRUE;

    return s1perms;
```

Library pseudocode for aarch64/translation/vmsa_faults/AArch64.S1HasAlignmentFault

```
// AArch64.S1HasAlignmentFault()
// ================================
// Returns whether stage 1 output fails alignment requirement on data accesses
// to Device memory

boolean AArch64.S1HasAlignmentFault(AccessDescriptor accdesc, boolean aligned,
                                    bit ntlsmd, MemoryAttributes memattrs)
    if accdesc.acctype == AccessType_IFETCH then
        return FALSE;
    elsif HaveMTEExt() && accdesc.tagaccess && accdesc.write then
        return (memattrs.memtype == MemType_Device &&
                ConstrainUnpredictable(Unpredictable_DEVICETAGSTORE) == Constraint_FAULT);
    elsif accdesc.a32lsmd && ntlsmd == '0' then
        return memattrs.memtype == MemType_Device &&  memattrs.device != DeviceType_GRE;
    elsif accdesc.acctype == AccessType_DCZero then
        return memattrs.memtype == MemType_Device;
    else
        return memattrs.memtype == MemType_Device && !aligned;
```

Library pseudocode for aarch64/translation/vmsa_faults/AArch64.S1IndirectBasePermissions

```
// AArch64.S1IndirectBasePermissions()
// ===================================
// Computes the stage 1 indirect base permissions

S1AccessControls AArch64.S1IndirectBasePermissions(Regime regime, TTWState walkstate,
                                                   S1TTWParams walkparams,
                                                   AccessDescriptor accdesc)

    bit  r,  w,  x,  gcs,  wxn,   overlay;
    bit pr, pw, px, pgcs, pwxn, p_overlay;
    bit ur, uw, ux, ugcs, uwxn, u_overlay;
    Permissions permissions = walkstate.permissions;
    S1AccessControls s1perms;

    // Apply privileged indirect permissions
    case permissions.ppi of
        when '0000' (pr,pw,px,pgcs) = ('0','0','0','0'); // No access
        when '0001' (pr,pw,px,pgcs) = ('1','0','0','0'); // Privileged read
        when '0010' (pr,pw,px,pgcs) = ('0','0','1','0'); // Privileged execute
        when '0011' (pr,pw,px,pgcs) = ('1','0','1','0'); // Privileged read and execute
        when '0100' (pr,pw,px,pgcs) = ('0','0','0','0'); // Reserved
        when '0101' (pr,pw,px,pgcs) = ('1','1','0','0'); // Privileged read and write
        when '0110' (pr,pw,px,pgcs) = ('1','1','1','0'); // Privileged read, write and execute
        when '0111' (pr,pw,px,pgcs) = ('1','1','1','0'); // Privileged read, write and execute
        when '1000' (pr,pw,px,pgcs) = ('1','0','0','0'); // Privileged read
        when '1001' (pr,pw,px,pgcs) = ('1','0','0','1'); // Privileged read and gcs
        when '1010' (pr,pw,px,pgcs) = ('1','0','1','0'); // Privileged read and execute
        when '1011' (pr,pw,px,pgcs) = ('0','0','0','0'); // Reserved
        when '1100' (pr,pw,px,pgcs) = ('1','1','0','0'); // Privileged read and write
        when '1101' (pr,pw,px,pgcs) = ('0','0','0','0'); // Reserved
        when '1110' (pr,pw,px,pgcs) = ('1','1','1','0'); // Privileged read, write and execute
        when '1111' (pr,pw,px,pgcs) = ('0','0','0','0'); // Reserved

    p_overlay = NOT(permissions.ppi[3]);
    pwxn = if permissions.ppi == '0110' then '1' else '0';

    if HasUnprivileged(regime) then
        // Apply unprivileged indirect permissions
        case permissions.upi of
            when '0000' (ur,uw,ux,ugcs) = ('0','0','0','0'); // No access
            when '0001' (ur,uw,ux,ugcs) = ('1','0','0','0'); // Unprivileged read
            when '0010' (ur,uw,ux,ugcs) = ('0','0','1','0'); // Unprivileged execute
            when '0011' (ur,uw,ux,ugcs) = ('1','0','1','0'); // Unprivileged read and execute
            when '0100' (ur,uw,ux,ugcs) = ('0','0','0','0'); // Reserved
            when '0101' (ur,uw,ux,ugcs) = ('1','1','0','0'); // Unprivileged read and write
            when '0110' (ur,uw,ux,ugcs) = ('1','1','1','0'); // Unprivileged read, write and execute
            when '0111' (ur,uw,ux,ugcs) = ('1','1','1','0'); // Unprivileged read, write and execute
            when '1000' (ur,uw,ux,ugcs) = ('1','0','0','0'); // Unprivileged read
            when '1001' (ur,uw,ux,ugcs) = ('1','0','0','1'); // Unprivileged read and gcs
            when '1010' (ur,uw,ux,ugcs) = ('1','0','1','0'); // Unprivileged read and execute
            when '1011' (ur,uw,ux,ugcs) = ('0','0','0','0'); // Reserved
            when '1100' (ur,uw,ux,ugcs) = ('1','1','0','0'); // Unprivileged read and write
            when '1101' (ur,uw,ux,ugcs) = ('0','0','0','0'); // Reserved
            when '1110' (ur,uw,ux,ugcs) = ('1','1','1','0'); // Unprivileged read,write and execute
            when '1111' (ur,uw,ux,ugcs) = ('0','0','0','0'); // Reserved

        u_overlay = NOT(permissions.upi[3]);
        uwxn = if permissions.upi == '0110' then '1' else '0';

        // If the decoded permissions has either px or pgcs along with either uw or ugcs,
        // then all effective Stage 1 Base Permissions are set to 0
        if ((px == '1' || pgcs == '1') && (uw == '1' || ugcs == '1')) then
            (pr,pw,px,pgcs) = ('0','0','0','0');
            (ur,uw,ux,ugcs) = ('0','0','0','0');

        if HavePANExt() && accdesc.pan && !(regime == Regime_EL10 && walkparams.nv1 == '1') then
            if PSTATE.PAN == '1' && (permissions.upi != '0000') then
                (pr,pw) = ('0','0');

    if accdesc.el == EL0 then
```

```
        (r,w,x,gcs,wxn,overlay) = (ur,uw,ux,ugcs,uwxn,u_overlay);
    else
        (r,w,x,gcs,wxn,overlay) = (pr,pw,px,pgcs,pwxn,p_overlay);

    // Prevent execution from Non-secure space by PE in secure state if SIF is set
    if accdesc.ss == SS_Secure && walkstate.baseaddress.paspace == PAS_NonSecure then
        x = x AND NOT(walkparams.sif);
        gcs = '0';
    // Prevent execution from non-Root space by Root
    if accdesc.ss == SS_Root && walkstate.baseaddress.paspace != PAS_Root then
        x = '0';
        gcs = '0';
    // Prevent execution from non-Realm space by Realm EL2 and Realm EL2&0
    if (accdesc.ss == SS_Realm && regime IN {Regime_EL2, Regime_EL20} &&
        walkstate.baseaddress.paspace != PAS_Realm) then
        x = '0';
        gcs = '0';

    s1perms.r       = r;
    s1perms.w       = w;
    s1perms.x       = x;
    s1perms.gcs     = gcs;
    s1perms.wxn     = wxn;
    s1perms.overlay = overlay == '1';

    return s1perms;
```

Library pseudocode for aarch64/translation/vmsa_faults/AArch64.S1OverlayPermissions

```
// AArch64.S1OverlayPermissions()
// ==============================
// Computes the stage 1 overlay permissions

S1AccessControls AArch64.S1OverlayPermissions(Regime regime, TTWState walkstate,
                                              AccessDescriptor accdesc)

    bit  r,  w,  x;
    bit pr, pw, px;
    bit ur, uw, ux;
    Permissions permissions = walkstate.permissions;
    S1AccessControls s1overlay_perms;

    S1PORType por = AArch64.S1POR(regime);
    integer bit_index = 4 * UInt(permissions.po_index);
    bits(4) ppo = por<bit_index+3:bit_index>;

    // Apply privileged overlay permissions
    case ppo of
        when '0000' (pr,pw,px) = ('0','0','0'); // No access
        when '0001' (pr,pw,px) = ('1','0','0'); // Privileged read
        when '0010' (pr,pw,px) = ('0','0','1'); // Privileged execute
        when '0011' (pr,pw,px) = ('1','0','1'); // Privileged read and execute
        when '0100' (pr,pw,px) = ('0','1','0'); // Privileged write
        when '0101' (pr,pw,px) = ('1','1','0'); // Privileged read and write
        when '0110' (pr,pw,px) = ('0','1','1'); // Privileged write and execute
        when '0111' (pr,pw,px) = ('1','1','1'); // Privileged read, write and execute
        when '1xxx' (pr,pw,px) = ('0','0','0'); // Reserved

    if HasUnprivileged(regime) then
        bits(4) upo = POR_EL0<bit_index+3:bit_index>;

        // Apply unprivileged overlay permissions
        case upo of
            when '0000' (ur,uw,ux) = ('0','0','0'); // No access
            when '0001' (ur,uw,ux) = ('1','0','0'); // Unprivileged read
            when '0010' (ur,uw,ux) = ('0','0','1'); // Unprivileged execute
            when '0011' (ur,uw,ux) = ('1','0','1'); // Unprivileged read and execute
            when '0100' (ur,uw,ux) = ('0','1','0'); // Unprivileged write
            when '0101' (ur,uw,ux) = ('1','1','0'); // Unprivileged read and write
            when '0110' (ur,uw,ux) = ('0','1','1'); // Unprivileged write and execute
            when '0111' (ur,uw,ux) = ('1','1','1'); // Unprivileged read, write and execute
            when '1xxx' (ur,uw,ux) = ('0','0','0'); // Reserved

    (r,w,x) = if accdesc.el == EL0 then (ur,uw,ux) else (pr,pw,px);

    s1overlay_perms.or = r;
    s1overlay_perms.ow = w;
    s1overlay_perms.ox = x;

    return s1overlay_perms;
```

Library pseudocode for aarch64/translation/vmsa_faults/AArch64.S1TxSZFaults

```
// AArch64.S1TxSZFaults()
// ======================
// Detect whether configuration of stage 1 TxSZ field generates a fault

boolean AArch64.S1TxSZFaults(Regime regime, S1TTWParams walkparams)
    mintxsz = AArch64.S1MinTxSZ(regime, walkparams.d128, walkparams.ds, walkparams.tgx);
    maxtxsz = AArch64.MaxTxSZ(walkparams.tgx);

    if UInt(walkparams.txsz) < mintxsz then
        return (Have52BitVAExt() ||
                boolean IMPLEMENTATION_DEFINED "Fault on TxSZ value below minimum");
    if UInt(walkparams.txsz) > maxtxsz then
        return boolean IMPLEMENTATION_DEFINED "Fault on TxSZ value above maximum";

    return FALSE;
```

Library pseudocode for aarch64/translation/vmsa_faults/AArch64.S2CheckPermissions

```
// AArch64.S2CheckPermissions()
// ============================
// Verifies memory access with available permissions.

(FaultRecord, boolean) AArch64.S2CheckPermissions(FaultRecord fault_in, TTWState walkstate,
                                                  S2TTWParams walkparams, AddressDescriptor ipa,
                                                  AccessDescriptor accdesc)
    MemType memtype = walkstate.memattrs.memtype;
    Permissions permissions = walkstate.permissions;
    FaultRecord fault = fault_in;
    S2AccessControls s2perms = AArch64.S2ComputePermissions(permissions, walkparams, accdesc);

    bit  r,  w;
    bit or, ow;

    if accdesc.acctype == AccessType_TTW then
        r = s2perms.r_mmu;
        w = s2perms.w_mmu;
        or = s2perms.or_mmu;
        ow = s2perms.ow_mmu;
    elsif accdesc.rcw then
        r = s2perms.r_rcw;
        w = s2perms.w_rcw;
        or = s2perms.or_rcw;
        ow = s2perms.ow_rcw;
    else
        r = s2perms.r;
        w = s2perms.w;
        or = s2perms.or;
        ow = s2perms.ow;

    if accdesc.acctype == AccessType_TTW then
        if (accdesc.toplevel && accdesc.varange == VARange_LOWER &&
               ((walkparams.tl0 == '1' && s2perms.toplevel0 == '0') ||
               (walkparams.tl1 == '1' && s2perms.<toplevel1,toplevel0> == '10'))) then
            fault.statuscode = Fault_Permission;
            fault.toplevel   = TRUE;
        elsif (accdesc.toplevel && accdesc.varange == VARange_UPPER &&
               ((walkparams.tl1 == '1' && s2perms.toplevel1 == '0') ||
               (walkparams.tl0 == '1' && s2perms.<toplevel1,toplevel0> == '01'))) then
            fault.statuscode = Fault_Permission;
            fault.toplevel   = TRUE;
        elsif walkparams.ptw == '1' && memtype == MemType_Device then
            fault.statuscode = Fault_Permission;
        elsif s2perms.overlay && or == '0' then
            fault.statuscode = Fault_Permission;
            fault.overlay    = TRUE;
        elsif accdesc.write && s2perms.overlay && ow == '0' then
            fault.statuscode = Fault_Permission;
            fault.overlay    = TRUE;
        // Prevent translation table walks in Non-secure space by Realm state
        elsif accdesc.ss == SS_Realm && walkstate.baseaddress.paspace != PAS_Realm then
            fault.statuscode = Fault_Permission;
        elsif r == '0' then
            fault.statuscode = Fault_Permission;
        elsif accdesc.write && w == '0' then
            fault.statuscode = Fault_Permission;
        elsif (accdesc.write && !(walkparams.<ha,hd> == '11') && walkparams.s2pie == '1' &&
                permissions.s2dirty == '0') then
            fault.statuscode = Fault_Permission;
            fault.dirtybit   = TRUE;
    // Stage 2 Permission fault due to AssuredOnly check
    elsif ((walkstate.s2assuredonly == '1' && !ipa.s1assured) ||
           (walkstate.s2assuredonly != '1' && HaveGCS() && VTCR_EL2.GCSH == '1' &&
            accdesc.acctype == AccessType_GCS && accdesc.el != EL0)) then
        fault.statuscode  = Fault_Permission;
        fault.assuredonly = TRUE;
    elsif accdesc.acctype == AccessType_IFETCH then
        if s2perms.overlay && s2perms.ox == '0' then
            fault.statuscode = Fault_Permission;
```

```
            fault.overlay    = TRUE;
        elsif (memtype == MemType_Device &&
                ConstrainUnpredictable(Unpredictable_INSTRDEVICE) == Constraint_FAULT) then
            fault.statuscode = Fault_Permission;

        // Prevent execution from Non-secure space by Realm state
        elsif accdesc.ss == SS_Realm && walkstate.baseaddress.paspace != PAS_Realm then
            fault.statuscode = Fault_Permission;
        elsif s2perms.x == '0' then
            fault.statuscode = Fault_Permission;

    elsif accdesc.acctype == AccessType_DC then
        if accdesc.cacheop == CacheOp_Invalidate then
            if !ELUsingAArch32(EL1) && s2perms.overlay && ow == '0' then
                fault.statuscode = Fault_Permission;
                fault.overlay    = TRUE;
            if !ELUsingAArch32(EL1) && w == '0' then
                fault.statuscode = Fault_Permission;
        elsif !ELUsingAArch32(EL1) && accdesc.el == EL0 && s2perms.overlay && or == '0' then
            fault.statuscode = Fault_Permission;
            fault.overlay    = TRUE;
        elsif (walkparams.cmow == '1' &&
                accdesc.opscope == CacheOpScope_PoC &&
                accdesc.cacheop == CacheOp_CleanInvalidate &&
                s2perms.overlay && ow == '0') then
            fault.statuscode = Fault_Permission;
            fault.overlay    = TRUE;
        elsif !ELUsingAArch32(EL1) && accdesc.el == EL0 && r == '0' then
            fault.statuscode = Fault_Permission;
        elsif (walkparams.cmow == '1' &&
                accdesc.opscope == CacheOpScope_PoC &&
                accdesc.cacheop == CacheOp_CleanInvalidate &&
                w == '0') then
            fault.statuscode = Fault_Permission;

    elsif accdesc.acctype == AccessType_IC then
        if (!ELUsingAArch32(EL1) && accdesc.el == EL0 && s2perms.overlay && or == '0' &&
                boolean IMPLEMENTATION_DEFINED "Permission fault on EL0 IC_IVAU execution") then
            fault.statuscode = Fault_Permission;
            fault.overlay    = TRUE;
        elsif walkparams.cmow == '1' && s2perms.overlay && ow == '0' then
            fault.statuscode = Fault_Permission;
            fault.overlay    = TRUE;
        elsif (!ELUsingAArch32(EL1) && accdesc.el == EL0 &&  r == '0' &&
                boolean IMPLEMENTATION_DEFINED "Permission fault on EL0 IC_IVAU execution") then
            fault.statuscode = Fault_Permission;
        elsif walkparams.cmow == '1' && w == '0' then
            fault.statuscode = Fault_Permission;

    elsif accdesc.read && s2perms.overlay && or == '0' then
        fault.statuscode = Fault_Permission;
        fault.overlay    = TRUE;
        fault.write      = FALSE;
    elsif accdesc.write && s2perms.overlay && ow == '0' then
        fault.statuscode = Fault_Permission;
        fault.overlay    = TRUE;
        fault.write      = TRUE;
    elsif accdesc.read && r == '0' then
        fault.statuscode = Fault_Permission;
        fault.write      = FALSE;
    elsif accdesc.write && w == '0' then
        fault.statuscode = Fault_Permission;
        fault.write      = TRUE;
    elsif ((accdesc.tagaccess || accdesc.tagchecked) &&
            ipa.memattrs.tags == MemTag_AllocationTagged &&
            permissions.s2tag_na == '1') then
        fault.statuscode = Fault_Permission;
        fault.tagaccess  = TRUE;
        fault.write      = accdesc.tagaccess && accdesc.write;
    elsif (accdesc.write && !(walkparams.<ha,hd> == '11') && walkparams.s2pie == '1' &&
```

```
            permissions.s2dirty == '0') then
        fault.statuscode = Fault_Permission;
        fault.dirtybit   = TRUE;
        fault.write      = TRUE;

    // MRO* allows only RCW and MMU writes
    boolean mro;
    if s2perms.overlay then
        mro = (s2perms.<w,w_rcw,w_mmu> AND s2perms.<ow,ow_rcw,ow_mmu>) == '011';
    else
        mro = s2perms.<w,w_rcw,w_mmu> == '011';

    return (fault, mro);
```

Library pseudocode for aarch64/translation/vmsa_faults/AArch64.S2ComputePermissions

```
// AArch64.S2ComputePermissions()
// ==============================
// Compute the overall stage 2 permissions.

S2AccessControls AArch64.S2ComputePermissions(Permissions permissions, S2TTWParams walkparams,
                                              AccessDescriptor accdesc)

    S2AccessControls s2perms;

    if walkparams.s2pie == '1' then
        s2perms = AArch64.S2IndirectBasePermissions(permissions, accdesc);
        s2perms.overlay = HaveS2POExt() && VTCR_EL2.S2POE == '1';
        if s2perms.overlay then
            s2overlay_perms = AArch64.S2OverlayPermissions(permissions, accdesc);
            s2perms.or          = s2overlay_perms.or;
            s2perms.ow          = s2overlay_perms.ow;
            s2perms.ox          = s2overlay_perms.ox;
            s2perms.or_rcw      = s2overlay_perms.or_rcw;
            s2perms.ow_rcw      = s2overlay_perms.ow_rcw;
            s2perms.or_mmu      = s2overlay_perms.or_mmu;
            s2perms.ow_mmu      = s2overlay_perms.ow_mmu;

            // Toplevel is applicable only when the effective S2 permissions is MRO
            if ((s2perms.<w,w_rcw,w_mmu> AND s2perms.<ow,ow_rcw,ow_mmu>) == '011') then
                s2perms.toplevel0 = s2perms.toplevel0 OR s2overlay_perms.toplevel0;
                s2perms.toplevel1 = s2perms.toplevel1 OR s2overlay_perms.toplevel1;

            else
                s2perms.toplevel0 = '0';
                s2perms.toplevel1 = '0';
    else
        s2perms = AArch64.S2DirectBasePermissions(permissions, accdesc);

    return s2perms;
```

Library pseudocode for aarch64/translation/vmsa_faults/AArch64.S2DirectBasePermissions

```
// AArch64.S2DirectBasePermissions()
// =================================
// Computes the stage 2 direct base permissions.

S2AccessControls AArch64.S2DirectBasePermissions(Permissions permissions,
                                                 AccessDescriptor accdesc)
    S2AccessControls s2perms;
    r = permissions.s2ap<0>;
    w = permissions.s2ap<1>;
    bit px, ux;
    case (permissions.s2xn:permissions.s2xnx) of
        when '00' (px,ux) = ('1','1');
        when '01' (px,ux) = ('0','1');
        when '10' (px,ux) = ('0','0');
        when '11' (px,ux) = ('1','0');

    x = if accdesc.el == EL0 then ux else px;
    s2perms.r = r;
    s2perms.w = w;
    s2perms.x = x;
    s2perms.r_rcw = r;
    s2perms.w_rcw = w;
    s2perms.r_mmu = r;
    s2perms.w_mmu = w;

    return s2perms;
```

Library pseudocode for aarch64/translation/vmsa_faults/AArch64.S2HasAlignmentFault

```
// AArch64.S2HasAlignmentFault()
// =============================
// Returns whether stage 2 output fails alignment requirement on data accesses
// to Device memory

boolean AArch64.S2HasAlignmentFault(AccessDescriptor accdesc, boolean aligned,
                                    MemoryAttributes memattrs)
    if accdesc.acctype == AccessType_IFETCH then
        return FALSE;
    elsif HaveMTEExt() && accdesc.tagaccess && accdesc.write then
        return (memattrs.memtype == MemType_Device &&
                ConstrainUnpredictable(Unpredictable_DEVICETAGSTORE) == Constraint_FAULT);
    elsif accdesc.acctype == AccessType_DCZero then
        return memattrs.memtype == MemType_Device;
    else
        return memattrs.memtype == MemType_Device && !aligned;
```

Library pseudocode for aarch64/translation/vmsa_faults/AArch64.S2InconsistentSL

```
// AArch64.S2InconsistentSL()
// ==========================
// Detect inconsistent configuration of stage 2 TxSZ and SL fields

boolean AArch64.S2InconsistentSL(S2TTWParams walkparams)
    startlevel   = AArch64.S2StartLevel(walkparams);
    levels       = FINAL_LEVEL - startlevel;
    granulebits  = TGxGranuleBits(walkparams.tgx);
    descsizelog2 = 3;
    stride       = granulebits - descsizelog2;

    // Input address size must at least be large enough to be resolved from the start level
    sl_min_iasize = (
        levels * stride // Bits resolved by table walk, except initial level
        + granulebits   // Bits directly mapped to output address
        + 1);           // At least 1 more bit to be decoded by initial level

    // Can accomodate 1 more stride in the level + concatenation of up to 2^4 tables
    sl_max_iasize = sl_min_iasize + (stride-1) + 4;
    // Configured Input Address size
    iasize        = AArch64.IASize(walkparams.txsz);

    return iasize < sl_min_iasize || iasize > sl_max_iasize;
```

Library pseudocode for aarch64/translation/vmsa_faults/AArch64.S2IndirectBasePermissions

```
// AArch64.S2IndirectBasePermissions()
// ===================================
// Computes the stage 2 indirect base permissions.

S2AccessControls AArch64.S2IndirectBasePermissions(Permissions permissions,
                                                   AccessDescriptor accdesc)
    bit r, w;
    bit r_rcw, w_rcw;
    bit r_mmu, w_mmu;
    bit px, ux;
    bit toplevel0, toplevel1;
    S2AccessControls s2perms;

    bits(4) s2pi = permissions.s2pi;
    case s2pi of
        when '0000' (r,w,px,ux,w_rcw,w_mmu) = ('0','0','0','0','0','0');  // No Access
        when '0001' (r,w,px,ux,w_rcw,w_mmu) = ('0','0','0','0','0','0');  // Reserved
        when '0010' (r,w,px,ux,w_rcw,w_mmu) = ('1','0','0','0','1','1');  // MRO
        when '0011' (r,w,px,ux,w_rcw,w_mmu) = ('1','0','0','0','1','1');  // MRO-TL1
        when '0100' (r,w,px,ux,w_rcw,w_mmu) = ('0','1','0','0','0','0');  // Write Only
        when '0101' (r,w,px,ux,w_rcw,w_mmu) = ('0','0','0','0','0','0');  // Reserved
        when '0110' (r,w,px,ux,w_rcw,w_mmu) = ('1','0','0','0','1','1');  // MRO-TL0
        when '0111' (r,w,px,ux,w_rcw,w_mmu) = ('1','0','0','0','1','1');  // MRO-TL01
        when '1000' (r,w,px,ux,w_rcw,w_mmu) = ('1','0','0','0','0','0');  // Read Only
        when '1001' (r,w,px,ux,w_rcw,w_mmu) = ('1','0','0','1','0','0');  // Read, Unpriv Execute
        when '1010' (r,w,px,ux,w_rcw,w_mmu) = ('1','0','1','0','0','0');  // Read, Priv Execute
        when '1011' (r,w,px,ux,w_rcw,w_mmu) = ('1','0','1','1','0','0');  // Read, All Execute
        when '1100' (r,w,px,ux,w_rcw,w_mmu) = ('1','1','0','0','1','1');  // RW
        when '1101' (r,w,px,ux,w_rcw,w_mmu) = ('1','1','0','1','1','1');  // RW, Unpriv Execute
        when '1110' (r,w,px,ux,w_rcw,w_mmu) = ('1','1','1','0','1','1');  // RW, Priv Execute
        when '1111' (r,w,px,ux,w_rcw,w_mmu) = ('1','1','1','1','1','1');  // RW, All Execute

    x = if accdesc.el == EL0 then ux else px;

    // RCW and MMU read permissions.
    (r_rcw, r_mmu) = (r, r);

    // Stage 2 Top Level Permission Attributes.
    case s2pi of
        when '0110' (toplevel0,toplevel1) = ('1','0');
        when '0011' (toplevel0,toplevel1) = ('0','1');
        when '0111' (toplevel0,toplevel1) = ('1','1');
        otherwise   (toplevel0,toplevel1) = ('0','0');

    s2perms.r = r;
    s2perms.w = w;
    s2perms.x = x;
    s2perms.r_rcw = r_rcw;
    s2perms.r_mmu = r_mmu;
    s2perms.w_rcw = w_rcw;
    s2perms.w_mmu = w_mmu;
    s2perms.toplevel0 = toplevel0;
    s2perms.toplevel1 = toplevel1;

    return s2perms;
```

Library pseudocode for aarch64/translation/vmsa_faults/AArch64.S2InvalidSL

```
// AArch64.S2InvalidSL()
// =====================
// Detect invalid configuration of SL field

boolean AArch64.S2InvalidSL(S2TTWParams walkparams)
    case walkparams.tgx of
        when TGx_4KB
            case walkparams.sl2:walkparams.sl0 of
                when '1x1' return TRUE;
                when '11x' return TRUE;
                when '010' return AArch64.PAMax() < 44;
                when '011' return !HaveSmallTranslationTableExt();
                otherwise  return FALSE;
        when TGx_16KB
            case walkparams.sl0 of
                when '11' return walkparams.ds == '0';
                when '10' return AArch64.PAMax() < 42;
                otherwise  return FALSE;
        when TGx_64KB
            case walkparams.sl0 of
                when '11'  return TRUE;
                when '10'  return AArch64.PAMax() < 44;
                otherwise  return FALSE;
```

Library pseudocode for aarch64/translation/vmsa_faults/AArch64.S2OverlayPermissions

```
// AArch64.S2OverlayPermissions()
// ==============================
// Computes the stage 2 overlay permissions.

S2AccessControls AArch64.S2OverlayPermissions(Permissions permissions, AccessDescriptor accdesc)
    bit r, w;
    bit r_rcw, w_rcw;
    bit r_mmu, w_mmu;
    bit px, ux;
    bit toplevel0, toplevel1;
    S2AccessControls s2overlay_perms;

    integer index = 4 * UInt(permissions.s2po_index);
    bits(4) s2po = S2POR_EL1[index+3 : index];
    case s2po of
        when '0000' (r,w,px,ux,w_rcw,w_mmu) = ('0','0','0','0','0','0');  // No Access
        when '0001' (r,w,px,ux,w_rcw,w_mmu) = ('0','0','0','0','0','0');  // Reserved
        when '0010' (r,w,px,ux,w_rcw,w_mmu) = ('1','0','0','0','1','1');  // MRO
        when '0011' (r,w,px,ux,w_rcw,w_mmu) = ('1','0','0','0','1','1');  // MRO-TL1
        when '0100' (r,w,px,ux,w_rcw,w_mmu) = ('0','1','0','0','0','0');  // Write Only
        when '0101' (r,w,px,ux,w_rcw,w_mmu) = ('0','0','0','0','0','0');  // Reserved
        when '0110' (r,w,px,ux,w_rcw,w_mmu) = ('1','0','0','0','1','1');  // MRO-TL0
        when '0111' (r,w,px,ux,w_rcw,w_mmu) = ('1','0','0','0','1','1');  // MRO-TL01
        when '1000' (r,w,px,ux,w_rcw,w_mmu) = ('1','0','0','0','0','0');  // Read Only
        when '1001' (r,w,px,ux,w_rcw,w_mmu) = ('1','0','0','1','0','0');  // Read, Unpriv Execute
        when '1010' (r,w,px,ux,w_rcw,w_mmu) = ('1','0','1','0','0','0');  // Read, Priv Execute
        when '1011' (r,w,px,ux,w_rcw,w_mmu) = ('1','0','1','1','0','0');  // Read, All Execute
        when '1100' (r,w,px,ux,w_rcw,w_mmu) = ('1','1','0','0','1','1');  // RW
        when '1101' (r,w,px,ux,w_rcw,w_mmu) = ('1','1','0','1','1','1');  // RW, Unpriv Execute
        when '1110' (r,w,px,ux,w_rcw,w_mmu) = ('1','1','1','0','1','1');  // RW, Priv Execute
        when '1111' (r,w,px,ux,w_rcw,w_mmu) = ('1','1','1','1','1','1');  // RW, All Execute

    x = if accdesc.el == EL0 then ux else px;

    // RCW and MMU read permissions.
    (r_rcw, r_mmu) = (r, r);

    // Stage 2 Top Level Permission Attributes.
    case s2po of
        when '0110' (toplevel0,toplevel1) = ('1','0');
        when '0011' (toplevel0,toplevel1) = ('0','1');
        when '0111' (toplevel0,toplevel1) = ('1','1');
        otherwise   (toplevel0,toplevel1) = ('0','0');

    s2overlay_perms.or = r;
    s2overlay_perms.ow = w;
    s2overlay_perms.ox = x;
    s2overlay_perms.or_rcw = r_rcw;
    s2overlay_perms.ow_rcw = w_rcw;
    s2overlay_perms.or_mmu = r_mmu;
    s2overlay_perms.ow_mmu = w_mmu;
    s2overlay_perms.toplevel0 = toplevel0;
    s2overlay_perms.toplevel1 = toplevel1;

    return s2overlay_perms;
```

Library pseudocode for aarch64/translation/vmsa_faults/AArch64.S2TxSZFaults

```
// AArch64.S2TxSZFaults()
// ======================
// Detect whether configuration of stage 2 TxSZ field generates a fault

boolean AArch64.S2TxSZFaults(S2TTWParams walkparams, boolean s1aarch64)
    mintxsz = AArch64.S2MinTxSZ(walkparams.d128, walkparams.ds, walkparams.tgx, s1aarch64);
    maxtxsz = AArch64.MaxTxSZ(walkparams.tgx);

    if UInt(walkparams.txsz) < mintxsz then
        return (Have52BitPAExt() ||
                boolean IMPLEMENTATION_DEFINED "Fault on TxSZ value below minimum");
    if UInt(walkparams.txsz) > maxtxsz then
        return boolean IMPLEMENTATION_DEFINED "Fault on TxSZ value above maximum";

    return FALSE;
```

Library pseudocode for aarch64/translation/vmsa_faults/AArch64.VAIsOutOfRange

```
// AArch64.VAIsOutOfRange()
// ========================
// Check bits not resolved by translation are identical and of accepted value

boolean AArch64.VAIsOutOfRange(bits(64) va_in, AccessType acctype,
                               Regime regime, S1TTWParams walkparams)
    bits(64) va = va_in;

    addrtop = AArch64.AddrTop(walkparams.tbid, acctype, walkparams.tbi);

    // If the VA has a Logical Address Tag then the bits holding the Logical Address Tag are
    // ignored when checking if the address is out of range.
    if walkparams.mtx == '1' then
        va<59:56> = if AArch64.GetVARange(va) == VARange_UPPER then '1111' else '0000';

    // Input Address size
    iasize  = AArch64.IASize(walkparams.txsz);

    // The min value of TxSZ can be 8, with LVA3 implemented.
    // If TxSZ is set to 8 iasize becomes 64 - 8 = 56
    // If tbi is also set, addrtop becomes 55
    // Then the return statements check va<56:55>
    // The check here is to guard against this corner case.
    if addrtop < iasize then
        return FALSE;

    if HasUnprivileged(regime) then
        if AArch64.GetVARange(va) == VARange_LOWER then
            return !IsZero(va<addrtop:iasize>);
        else
            return !IsOnes(va<addrtop:iasize>);
    else
        return !IsZero(va<addrtop:iasize>);
```

Library pseudocode for aarch64/translation/vmsa_memattr/AArch64.S2ApplyFWBMemAttrs

```
// AArch64.S2ApplyFWBMemAttrs()
// ============================
// Apply stage 2 forced Write-Back on stage 1 memory attributes.

MemoryAttributes AArch64.S2ApplyFWBMemAttrs(MemoryAttributes s1_memattrs, S2TTWParams walkparams,
                                            bits(N) descriptor)
    MemoryAttributes memattrs;
    s2_attr = descriptor<5:2>;
    s2_sh   = if walkparams.ds == '1' then walkparams.sh else descriptor<9:8>;
    s2_fnxs = descriptor<11>;

    if s2_attr<2> == '0' then          // S2 Device, S1 any
        s2_device = DecodeDevice(s2_attr<1:0>);
        memattrs.memtype = MemType_Device;
        if s1_memattrs.memtype == MemType_Device then
            memattrs.device = S2CombineS1Device(s1_memattrs.device, s2_device);
        else
            memattrs.device = s2_device;

        memattrs.xs = s1_memattrs.xs;

    elsif s2_attr<1:0> == '11' then    // S2 attr = S1 attr
        memattrs = s1_memattrs;

    elsif s2_attr<1:0> == '10' then    // Force writeback
        memattrs.memtype = MemType_Normal;
        memattrs.inner.attrs = MemAttr_WB;
        memattrs.outer.attrs = MemAttr_WB;

        if (s1_memattrs.memtype == MemType_Normal &&
                s1_memattrs.inner.attrs != MemAttr_NC) then
            memattrs.inner.hints     = s1_memattrs.inner.hints;
            memattrs.inner.transient = s1_memattrs.inner.transient;
        else
            memattrs.inner.hints     = MemHint_RWA;
            memattrs.inner.transient = FALSE;

        if (s1_memattrs.memtype == MemType_Normal &&
                s1_memattrs.outer.attrs != MemAttr_NC) then
            memattrs.outer.hints     = s1_memattrs.outer.hints;
            memattrs.outer.transient = s1_memattrs.outer.transient;
        else
            memattrs.outer.hints     = MemHint_RWA;
            memattrs.outer.transient = FALSE;

        memattrs.xs = '0';

    else                               // Non-cacheable unless S1 is device
        if s1_memattrs.memtype == MemType_Device then
            memattrs = s1_memattrs;
        else
            MemAttrHints cacheability_attr;
            cacheability_attr.attrs = MemAttr_NC;

            memattrs.memtype = MemType_Normal;
            memattrs.inner   = cacheability_attr;
            memattrs.outer   = cacheability_attr;

            memattrs.xs = s1_memattrs.xs;

    s2_shareability = DecodeShareability(s2_sh);
    memattrs.shareability = S2CombineS1Shareability(s1_memattrs.shareability, s2_shareability);
    memattrs.tags         = S2MemTagType(memattrs, s1_memattrs.tags);
    memattrs.notagaccess  = (s2_attr<3:1> == '111' && memattrs.tags == MemTag_AllocationTagged);

    if s2_fnxs == '1' then
        memattrs.xs = '0';

    memattrs.shareability = EffectiveShareability(memattrs);
    return memattrs;
```

Library pseudocode for aarch64/translation/vmsa_tlbcontext/AArch64.GetS1TLBContext

```
// AArch64.GetS1TLBContext()
// =========================
// Gather translation context for accesses with VA to match against TLB entries

TLBContext AArch64.GetS1TLBContext(Regime regime, SecurityState ss, bits(64) va, TGx tg)
    TLBContext tlbcontext;

    case regime of
        when Regime_EL3  tlbcontext = AArch64.TLBContextEL3(ss, va, tg);
        when Regime_EL2  tlbcontext = AArch64.TLBContextEL2(ss, va, tg);
        when Regime_EL20 tlbcontext = AArch64.TLBContextEL20(ss, va, tg);
        when Regime_EL10 tlbcontext = AArch64.TLBContextEL10(ss, va, tg);

    tlbcontext.includes_s1  = TRUE;
    // The following may be amended for EL1&0 Regime if caching of stage 2 is successful
    tlbcontext.includes_s2  = FALSE;
    // The following may be amended if Granule Protection Check passes
    tlbcontext.includes_gpt = FALSE;
    return tlbcontext;
```

Library pseudocode for aarch64/translation/vmsa_tlbcontext/AArch64.GetS2TLBContext

```
// AArch64.GetS2TLBContext()
// =========================
// Gather translation context for accesses with IPA to match against TLB entries

TLBContext AArch64.GetS2TLBContext(SecurityState ss, FullAddress ipa, TGx tg)
    assert EL2Enabled();

    TLBContext tlbcontext;

    tlbcontext.ss       = ss;
    tlbcontext.regime   = Regime_EL10;
    tlbcontext.ipaspace = ipa.paspace;
    tlbcontext.vmid     = VMID[];
    tlbcontext.tg       = tg;
    tlbcontext.ia       = ZeroExtend(ipa.address, 64);
    if HaveCommonNotPrivateTransExt() then
        tlbcontext.cnp = if ipa.paspace == PAS_Secure then VSTTBR_EL2.CnP else VTTBR_EL2.CnP;
    else
        tlbcontext.cnp = '0';

    tlbcontext.includes_s1  = FALSE;
    tlbcontext.includes_s2  = TRUE;
    // This amy be amended if Granule Protection Check passes
    tlbcontext.includes_gpt = FALSE;
    return tlbcontext;
```

Library pseudocode for aarch64/translation/vmsa_tlbcontext/AArch64.TLBContextEL10

```
// AArch64.TLBContextEL10()
// ========================
// Gather translation context for accesses under EL10 regime to match against TLB entries

TLBContext AArch64.TLBContextEL10(SecurityState ss, bits(64) va, TGx tg)
    TLBContext tlbcontext;

    tlbcontext.ss     = ss;
    tlbcontext.regime = Regime_EL10;
    tlbcontext.vmid   = VMID[];
    tlbcontext.asid   = if TCR_EL1.A1 == '0' then TTBR0_EL1.ASID else TTBR1_EL1.ASID;
    if TCR_EL1.AS == '0' then
        tlbcontext.asid<15:8> = Zeros(8);
    tlbcontext.tg     = tg;
    tlbcontext.ia     = va;

    if HaveCommonNotPrivateTransExt() then
        if AArch64.GetVARange(va) == VARange_LOWER then
            tlbcontext.cnp = TTBR0_EL1.CnP;
        else
            tlbcontext.cnp = TTBR1_EL1.CnP;
    else
        tlbcontext.cnp = '0';

    return tlbcontext;
```

Library pseudocode for aarch64/translation/vmsa_tlbcontext/AArch64.TLBContextEL2

```
// AArch64.TLBContextEL2()
// =======================
// Gather translation context for accesses under EL2 regime to match against TLB entries

TLBContext AArch64.TLBContextEL2(SecurityState ss, bits(64) va, TGx tg)
    TLBContext tlbcontext;

    tlbcontext.ss     = ss;
    tlbcontext.regime = Regime_EL2;
    tlbcontext.tg     = tg;
    tlbcontext.ia     = va;
    tlbcontext.cnp    = if HaveCommonNotPrivateTransExt() then TTBR0_EL2.CnP else '0';

    return tlbcontext;
```

Library pseudocode for aarch64/translation/vmsa_tlbcontext/AArch64.TLBContextEL20

```
// AArch64.TLBContextEL20()
// ========================
// Gather translation context for accesses under EL20 regime to match against TLB entries

TLBContext AArch64.TLBContextEL20(SecurityState ss, bits(64) va, TGx tg)
    TLBContext tlbcontext;

    tlbcontext.ss     = ss;
    tlbcontext.regime = Regime_EL20;
    tlbcontext.asid   = if TCR_EL2.A1 == '0' then TTBR0_EL2.ASID else TTBR1_EL2.ASID;
    if TCR_EL2.AS == '0' then
        tlbcontext.asid<15:8> = Zeros(8);
    tlbcontext.tg     = tg;
    tlbcontext.ia     = va;

    if HaveCommonNotPrivateTransExt() then
        if AArch64.GetVARange(va) == VARange_LOWER then
            tlbcontext.cnp = TTBR0_EL2.CnP;
        else
            tlbcontext.cnp = TTBR1_EL2.CnP;
    else
        tlbcontext.cnp = '0';

    return tlbcontext;
```

Library pseudocode for aarch64/translation/vmsa_tlbcontext/AArch64.TLBContextEL3

```
// AArch64.TLBContextEL3()
// =======================
// Gather translation context for accesses under EL3 regime to match against TLB entries

TLBContext AArch64.TLBContextEL3(SecurityState ss, bits(64) va, TGx tg)
    TLBContext tlbcontext;

    tlbcontext.ss     = ss;
    tlbcontext.regime = Regime_EL3;
    tlbcontext.tg     = tg;
    tlbcontext.ia     = va;
    tlbcontext.cnp    = if HaveCommonNotPrivateTransExt() then TTBR0_EL3.CnP else '0';

    return tlbcontext;
```

Library pseudocode for aarch64/translation/vmsa_translation/AArch64.FullTranslate

```
// AArch64.FullTranslate()
// =======================
// Address translation as specified by VMSA
// Alignment check NOT due to memory type is expected to be done before translation

AddressDescriptor AArch64.FullTranslate(bits(64) va, AccessDescriptor accdesc, boolean aligned)
    Regime regime = TranslationRegime(accdesc.el);
    FaultRecord fault = NoFault(accdesc);

    AddressDescriptor ipa;
    (fault, ipa) = AArch64.S1Translate(fault, regime, va, aligned, accdesc);

    if fault.statuscode != Fault_None then
        return CreateFaultyAddressDescriptor(va, fault);

    assert (accdesc.ss == SS_Realm) IMPLIES EL2Enabled();
    if regime == Regime_EL10 && EL2Enabled() then
        s1aarch64 = TRUE;
        AddressDescriptor pa;
        (fault, pa) = AArch64.S2Translate(fault, ipa, s1aarch64, aligned, accdesc);

        if fault.statuscode != Fault_None then
            return CreateFaultyAddressDescriptor(va, fault);
        else
            return pa;
    else
        return ipa;
```

Library pseudocode for aarch64/translation/vmsa_translation/AArch64.MemSwapTableDesc

```
// AArch64.MemSwapTableDesc()
// ==========================
// Perform HW update of table descriptor as an atomic operation

(FaultRecord, bits(N)) AArch64.MemSwapTableDesc(FaultRecord fault_in, bits(N) prev_desc,
                                                bits(N) new_desc, bit ee,
                                                AccessDescriptor  descaccess,
                                                AddressDescriptor descpaddr)
    FaultRecord fault = fault_in;
    boolean iswrite;

    if HaveRME() then
        fault.gpcf = GranuleProtectionCheck(descpaddr, descaccess);
        if fault.gpcf.gpf != GPCF_None then
            fault.statuscode = Fault_GPCFOnWalk;
            fault.paddress   = descpaddr.paddress;
            fault.gpcfs2walk = fault.secondstage;
            return (fault, bits(N) UNKNOWN);

    // All observers in the shareability domain observe the
    // following memory read and write accesses atomically.
    bits(N) mem_desc;
    PhysMemRetStatus memstatus;
    (memstatus, mem_desc) = PhysMemRead(descpaddr, N DIV 8, descaccess);

    if ee == '1' then
        mem_desc = BigEndianReverse(mem_desc);

    if IsFault(memstatus) then
        iswrite = FALSE;
        fault = HandleExternalTTWAbort(memstatus, iswrite, descpaddr, descaccess, N DIV 8, fault);
        if IsFault(fault.statuscode) then
            return (fault, bits(N) UNKNOWN);

    if mem_desc == prev_desc  then
        ordered_new_desc = if ee == '1' then BigEndianReverse(new_desc) else new_desc;
        memstatus = PhysMemWrite(descpaddr, N DIV 8, descaccess, ordered_new_desc);

        if IsFault(memstatus) then
            iswrite = TRUE;
            fault = HandleExternalTTWAbort(memstatus, iswrite, descpaddr, descaccess, N DIV 8,
                                           fault);

            if IsFault(fault.statuscode) then
                return (fault, bits(N) UNKNOWN);

        // Reflect what is now in memory (in little endian format)
        mem_desc = new_desc;

    return (fault, mem_desc);
```

Library pseudocode for aarch64/translation/vmsa_translation/AArch64.S1DisabledOutput

```
// AArch64.S1DisabledOutput()
// ==========================
// Map the VA to IPA/PA and assign default memory attributes

(FaultRecord, AddressDescriptor) AArch64.S1DisabledOutput(FaultRecord fault_in, Regime regime,
                                                          bits(64) va_in, AccessDescriptor accdesc,
                                                          boolean aligned)

    bits(64) va = va_in;
    walkparams = AArch64.GetS1TTWParams(regime, accdesc.ss, va);
    FaultRecord fault = fault_in;

    // No memory page is guarded when stage 1 address translation is disabled
    SetInGuardedPage(FALSE);

    // Output Address
    FullAddress oa;
    oa.address = va<55:0>;
    case accdesc.ss of
        when SS_Secure    oa.paspace = PAS_Secure;
        when SS_NonSecure oa.paspace = PAS_NonSecure;
        when SS_Root      oa.paspace = PAS_Root;
        when SS_Realm     oa.paspace = PAS_Realm;

    MemoryAttributes memattrs;
    if regime == Regime_EL10 && EL2Enabled() && walkparams.dc == '1' then
        MemAttrHints default_cacheability;
        default_cacheability.attrs     = MemAttr_WB;
        default_cacheability.hints     = MemHint_RWA;
        default_cacheability.transient = FALSE;

        memattrs.memtype      = MemType_Normal;
        memattrs.outer        = default_cacheability;
        memattrs.inner        = default_cacheability;
        memattrs.shareability = Shareability_NSH;
        if walkparams.dct == '1' then
            memattrs.tags     = MemTag_AllocationTagged;
        elsif walkparams.mtx == '1' then
            memattrs.tags     = MemTag_CanonicallyTagged;
            if walkparams.tbi == '0' then
                // For the purpose of the checks in this function, the MTE tag bits are ignored.
                va<59:56>     = Replicate(va<55>, 4);
        else
            memattrs.tags     = MemTag_Untagged;
        memattrs.xs           = '0';
    elsif accdesc.acctype == AccessType_IFETCH then
        MemAttrHints i_cache_attr;
        if AArch64.S1ICacheEnabled(regime) then
            i_cache_attr.attrs     = MemAttr_WT;
            i_cache_attr.hints     = MemHint_RA;
            i_cache_attr.transient = FALSE;
        else
            i_cache_attr.attrs     = MemAttr_NC;

        memattrs.memtype      = MemType_Normal;
        memattrs.outer        = i_cache_attr;
        memattrs.inner        = i_cache_attr;
        memattrs.shareability = Shareability_OSH;
        if walkparams.mtx == '1' then
            memattrs.tags     = MemTag_CanonicallyTagged;
        else
            memattrs.tags     = MemTag_Untagged;
        memattrs.xs           = '1';
    else
        memattrs.memtype      = MemType_Device;
        memattrs.device       = DeviceType_nGnRnE;
        memattrs.shareability = Shareability_OSH;
        if walkparams.mtx == '1' then
            memattrs.tags = MemTag_CanonicallyTagged;
            if walkparams.tbi == '0' then
```

```
                // For the purpose of the checks in this function, the MTE tag bits are ignored.
                if HasUnprivileged(regime) then
                    va<59:56> = Replicate(va<55>, 4);
                else
                    va<59:56> = '0000';
        else
            memattrs.tags = MemTag_Untagged;
        memattrs.xs           = '1';
    memattrs.notagaccess = FALSE;

    fault.level = 0;
    addrtop     = AArch64.AddrTop(walkparams.tbid, accdesc.acctype, walkparams.tbi);

    if !IsZero(va<addrtop:AArch64.PAMax()>) then
        fault.statuscode = Fault_AddressSize;
    elsif AArch64.S1HasAlignmentFault(accdesc, aligned, walkparams.ntlsmd, memattrs) then
        fault.statuscode = Fault_Alignment;

    if fault.statuscode != Fault_None then
        return (fault, AddressDescriptor UNKNOWN);
    else
        ipa = CreateAddressDescriptor(va_in, oa, memattrs);
        ipa.mecid = AArch64.S1DisabledOutputMECID(walkparams, regime, ipa.paddress.paspace);
        return (fault, ipa);
```

Library pseudocode for aarch64/translation/vmsa_translation/AArch64.S1Translate

```
// AArch64.S1Translate()
// =====================
// Translate VA to IPA/PA depending on the regime

(FaultRecord, AddressDescriptor) AArch64.S1Translate(FaultRecord fault_in, Regime regime,
                                                     bits(64) va, boolean aligned,
                                                     AccessDescriptor accdesc)
    FaultRecord fault = fault_in;
    // Prepare fault fields in case a fault is detected
    fault.secondstage = FALSE;
    fault.s2fs1walk   = FALSE;

    if !AArch64.S1Enabled(regime, accdesc.acctype) then
        return AArch64.S1DisabledOutput(fault, regime, va, accdesc, aligned);

    walkparams = AArch64.GetS1TTWParams(regime, accdesc.ss, va);

    constant integer s1mintxsz = AArch64.S1MinTxSZ(regime, walkparams.d128,
                                                   walkparams.ds, walkparams.tgx);
    constant integer s1maxtxsz = AArch64.MaxTxSZ(walkparams.tgx);
    if AArch64.S1TxSZFaults(regime, walkparams) then
        fault.statuscode = Fault_Translation;
        fault.level      = 0;
        return (fault, AddressDescriptor UNKNOWN);
    elsif UInt(walkparams.txsz) < s1mintxsz then
        walkparams.txsz = s1mintxsz<5:0>;
    elsif UInt(walkparams.txsz) > s1maxtxsz then
        walkparams.txsz = s1maxtxsz<5:0>;

    if AArch64.VAIsOutOfRange(va, accdesc.acctype, regime, walkparams) then
        fault.statuscode = Fault_Translation;
        fault.level      = 0;
        return (fault, AddressDescriptor UNKNOWN);

    if accdesc.el == EL0 && walkparams.e0pd == '1' then
        fault.statuscode = Fault_Translation;
        fault.level      = 0;
        return (fault, AddressDescriptor UNKNOWN);

    if HaveTME() && accdesc.el == EL0 && walkparams.nfd == '1' && accdesc.transactional then
        fault.statuscode = Fault_Translation;
        fault.level      = 0;
        return (fault, AddressDescriptor UNKNOWN);

    if HaveSVE() && accdesc.el == EL0 && walkparams.nfd == '1' && (
            (accdesc.nonfault && accdesc.contiguous) ||
            (accdesc.firstfault && !accdesc.first && !accdesc.contiguous)) then
        fault.statuscode = Fault_Translation;
        fault.level      = 0;
        return (fault, AddressDescriptor UNKNOWN);

    AddressDescriptor descipaddr;
    TTWState walkstate;
    bits(128) descriptor;
    bits(128) new_desc;
    bits(128) mem_desc;
    repeat
        if walkparams.d128 == '1' then
            (fault, descipaddr, walkstate, descriptor) = AArch64.S1Walk(fault, walkparams, va,
                                                                        regime, accdesc, 128);
        else
            (fault, descipaddr, walkstate, descriptor<63:0>) = AArch64.S1Walk(fault, walkparams,
                                                                              va, regime, accdesc,
                                                                              64);
            descriptor<127:64> = Zeros(64);
        if fault.statuscode != Fault_None then
            return (fault, AddressDescriptor UNKNOWN);

        if accdesc.acctype == AccessType_IFETCH then
            // Flag the fetched instruction is from a guarded page
```

```
            SetInGuardedPage(walkstate.guardedpage == '1');

        if AArch64.S1HasAlignmentFault(accdesc, aligned, walkparams.ntlsmd,
                                        walkstate.memattrs) then
            fault.statuscode = Fault_Alignment;

        if fault.statuscode == Fault_None then
            fault = AArch64.S1CheckPermissions(fault, regime, walkstate, walkparams, accdesc);

        new_desc = descriptor;
        if walkparams.ha == '1' && AArch64.SettingAccessFlagPermitted(fault) then
            // Set descriptor AF bit
            new_desc<10> = '1';

        // If HW update of dirty bit is enabled, the walk state permissions
        // will already reflect a configuration permitting writes.
        // The update of the descriptor occurs only if the descriptor bits in
        // memory do not reflect that and the access instigates a write.

        if (AArch64.SettingDirtyStatePermitted(fault) &&
                walkparams.ha  == '1' &&
                walkparams.hd  == '1' &&
                (walkparams.pie == '1' || descriptor<51> == '1') &&
                accdesc.write &&
                !(accdesc.acctype IN {AccessType_AT, AccessType_IC, AccessType_DC})) then
            // Clear descriptor AP[2]/nDirty bit permitting stage 1 writes
            new_desc<7> = '0';

        // Either the access flag was clear or AP[2]/nDirty is set
        if new_desc != descriptor then
            AddressDescriptor descpaddr;
            descaccess = CreateAccDescTTEUpdate(accdesc);
            if regime == Regime_EL10 && EL2Enabled() then
                FaultRecord s2fault;
                s1aarch64 = TRUE;
                s2aligned = TRUE;
                (s2fault, descpaddr) = AArch64.S2Translate(fault, descipaddr, s1aarch64, s2aligned,
                                                           descaccess);

                if s2fault.statuscode != Fault_None then
                    return (s2fault, AddressDescriptor UNKNOWN);

            else
                descpaddr = descipaddr;
            if walkparams.d128 == '1' then
                (fault, mem_desc) = AArch64.MemSwapTableDesc(fault, descriptor, new_desc,
                                                             walkparams.ee, descaccess, descpaddr);
            else
                (fault, mem_desc<63:0>) = AArch64.MemSwapTableDesc(fault, descriptor<63:0>,
                                                                   new_desc<63:0>, walkparams.ee,
                                                                   descaccess, descpaddr);
                mem_desc<127:64> = Zeros(64);

    until new_desc == descriptor || mem_desc == new_desc;

    if fault.statuscode != Fault_None then
        return (fault, AddressDescriptor UNKNOWN);

    // Output Address
    oa = StageOA(va, walkparams.d128, walkparams.tgx, walkstate);
    MemoryAttributes memattrs;
    if (accdesc.acctype == AccessType_IFETCH &&
        (walkstate.memattrs.memtype == MemType_Device || !AArch64.S1ICacheEnabled(regime))) then
        // Treat memory attributes as Normal Non-Cacheable
        memattrs = NormalNCMemAttr();
        memattrs.xs = walkstate.memattrs.xs;
    elsif (accdesc.acctype != AccessType_IFETCH && !AArch64.S1DCacheEnabled(regime) &&
             walkstate.memattrs.memtype == MemType_Normal) then
        // Treat memory attributes as Normal Non-Cacheable
        memattrs = NormalNCMemAttr();
```

```
        memattrs.xs = walkstate.memattrs.xs;

        // The effect of SCTLR_ELx.C when '0' is Constrained UNPREDICTABLE
        // on the Tagged attribute
        if (HaveMTE2Ext() && walkstate.memattrs.tags == MemTag_AllocationTagged &&
            !ConstrainUnpredictableBool(Unpredictable_S1CTAGGED)) then
            memattrs.tags = MemTag_Untagged;
    else
        memattrs = walkstate.memattrs;

    // Shareability value of stage 1 translation subject to stage 2 is IMPLEMENTATION DEFINED
    // to be either effective value or descriptor value
    if (regime == Regime_EL10 && EL2Enabled() && HCR_EL2.VM == '1' &&
            !(boolean IMPLEMENTATION_DEFINED "Apply effective shareability at stage 1")) then
        memattrs.shareability = walkstate.memattrs.shareability;
    else
        memattrs.shareability = EffectiveShareability(memattrs);

    if accdesc.ls64 && memattrs.memtype == MemType_Normal then
        if memattrs.inner.attrs != MemAttr_NC || memattrs.outer.attrs != MemAttr_NC then
            fault.statuscode = Fault_Exclusive;
            return (fault, AddressDescriptor UNKNOWN);

    ipa = CreateAddressDescriptor(va, oa, memattrs);
    ipa.s1assured = walkstate.s1assured;
    varange   = AArch64.GetVARange(va);
    ipa.mecid = AArch64.S1OutputMECID(walkparams, regime, varange, ipa.paddress.paspace,
                                      descriptor);
    return (fault, ipa);
```

Library pseudocode for aarch64/translation/vmsa_translation/AArch64.S2Translate

```
// AArch64.S2Translate()
// =====================
// Translate stage 1 IPA to PA and combine memory attributes

(FaultRecord, AddressDescriptor) AArch64.S2Translate(FaultRecord fault_in, AddressDescriptor ipa,
                                                     boolean s1aarch64, boolean aligned,
                                                     AccessDescriptor accdesc)

    walkparams = AArch64.GetS2TTWParams(accdesc.ss, ipa.paddress.paspace, s1aarch64);
    FaultRecord fault = fault_in;
    boolean s2fs1mro;

    // Prepare fault fields in case a fault is detected
    fault.statuscode  = Fault_None; // Ignore any faults from stage 1
    fault.secondstage = TRUE;
    fault.s2fs1walk   = accdesc.acctype == AccessType_TTW;
    fault.ipaddress   = ipa.paddress;

    if walkparams.vm != '1' then
        // Stage 2 translation is disabled
        return (fault, ipa);

    constant integer s2mintxsz = AArch64.S2MinTxSZ(walkparams.d128, walkparams.ds,
                                                   walkparams.tgx, s1aarch64);
    constant integer s2maxtxsz = AArch64.MaxTxSZ(walkparams.tgx);
    if AArch64.S2TxSZFaults(walkparams, s1aarch64) then
        fault.statuscode = Fault_Translation;
        fault.level      = 0;
        return (fault, AddressDescriptor UNKNOWN);
    elsif UInt(walkparams.txsz) < s2mintxsz then
        walkparams.txsz = s2mintxsz<5:0>;
    elsif UInt(walkparams.txsz) > s2maxtxsz then
        walkparams.txsz = s2maxtxsz<5:0>;

    if (walkparams.d128 == '0' &&
        (AArch64.S2InvalidSL(walkparams) || AArch64.S2InconsistentSL(walkparams))) then
        fault.statuscode = Fault_Translation;
        fault.level      = 0;
        return (fault, AddressDescriptor UNKNOWN);

    if AArch64.IPAIsOutOfRange(ipa.paddress.address, walkparams) then
        fault.statuscode = Fault_Translation;
        fault.level      = 0;
        return (fault, AddressDescriptor UNKNOWN);

    AddressDescriptor descpaddr;
    TTWState walkstate;
    bits(128) descriptor;
    bits(128) new_desc;
    bits(128) mem_desc;
    repeat
        if walkparams.d128 == '1' then
            (fault, descpaddr, walkstate, descriptor) = AArch64.S2Walk(fault, ipa, walkparams,
                                                                       accdesc, 128);
        else
            (fault, descpaddr, walkstate, descriptor<63:0>) = AArch64.S2Walk(fault, ipa,
                                                                             walkparams, accdesc,
                                                                             64);
            descriptor<127:64> = Zeros(64);
        if fault.statuscode != Fault_None then
            return (fault, AddressDescriptor UNKNOWN);

        if AArch64.S2HasAlignmentFault(accdesc, aligned, walkstate.memattrs) then
            fault.statuscode = Fault_Alignment;

        if fault.statuscode == Fault_None then
            (fault, s2fs1mro) = AArch64.S2CheckPermissions(fault, walkstate, walkparams, ipa,
                                                           accdesc);

        new_desc = descriptor;
```

```
        if walkparams.ha == '1' && AArch64.SettingAccessFlagPermitted(fault) then
            // Set descriptor AF bit
            new_desc<10> = '1';

        // If HW update of dirty bit is enabled, the walk state permissions
        // will already reflect a configuration permitting writes.
        // The update of the descriptor occurs only if the descriptor bits in
        // memory do not reflect that and the access instigates a write.

        if (AArch64.SettingDirtyStatePermitted(fault) &&
                walkparams.ha  == '1' &&
                walkparams.hd  == '1' &&
                (walkparams.s2pie == '1' || descriptor<51> == '1') &&
                accdesc.write &&
                !(accdesc.acctype IN {AccessType_AT, AccessType_IC, AccessType_DC})) then
            // Set descriptor S2AP[1]/Dirty bit permitting stage 2 writes
            new_desc<7> = '1';

        // Either the access flag was clear or S2AP[1]/Dirty is clear
        if new_desc != descriptor then
            AccessDescriptor descaccess = CreateAccDescTTEUpdate(accdesc);
            if walkparams.d128 == '1' then
                (fault, mem_desc) = AArch64.MemSwapTableDesc(fault, descriptor, new_desc,
                                                             walkparams.ee, descaccess,
                                                             descpaddr);
            else
                (fault, mem_desc<63:0>) = AArch64.MemSwapTableDesc(fault, descriptor<63:0>,
                                                                   new_desc<63:0>, walkparams.ee,
                                                                   descaccess, descpaddr);
                mem_desc<127:64> = Zeros(64);

    until new_desc == descriptor || mem_desc == new_desc;

    if fault.statuscode != Fault_None then
        return (fault, AddressDescriptor UNKNOWN);

    ipa_64 = ZeroExtend(ipa.paddress.address, 64);
    // Output Address
    oa = StageOA(ipa_64, walkparams.d128, walkparams.tgx, walkstate);
    MemoryAttributes s2_memattrs;
    if ((accdesc.acctype == AccessType_TTW &&
            walkstate.memattrs.memtype == MemType_Device && walkparams.ptw == '0') ||
        (accdesc.acctype == AccessType_IFETCH &&
            (walkstate.memattrs.memtype == MemType_Device || HCR_EL2.ID == '1')) ||
        (accdesc.acctype != AccessType_IFETCH &&
             walkstate.memattrs.memtype == MemType_Normal && HCR_EL2.CD == '1')) then
        // Treat memory attributes as Normal Non-Cacheable
        s2_memattrs = NormalNCMemAttr();
        s2_memattrs.xs = walkstate.memattrs.xs;
    else
        s2_memattrs = walkstate.memattrs;

    if accdesc.ls64 && s2_memattrs.memtype == MemType_Normal then
        if s2_memattrs.inner.attrs != MemAttr_NC || s2_memattrs.outer.attrs != MemAttr_NC then
            fault.statuscode = Fault_Exclusive;
            return (fault, AddressDescriptor UNKNOWN);

    s2aarch64 = TRUE;
    MemoryAttributes memattrs;
    if walkparams.fwb == '0' then
        memattrs = S2CombineS1MemAttrs(ipa.memattrs, s2_memattrs, s2aarch64);
    else
        memattrs = s2_memattrs;

    pa = CreateAddressDescriptor(ipa.vaddress, oa, memattrs);
    pa.s2fs1mro = s2fs1mro;
    pa.mecid = AArch64.S2OutputMECID(walkparams, pa.paddress.paspace, descriptor);
    return (fault, pa);
```

Library pseudocode for aarch64/translation/vmsa_translation/AArch64.SettingAccessFlagPermitted

```
// AArch64.SettingAccessFlagPermitted()
// ====================================
// Determine whether the access flag could be set by HW given the fault status

boolean AArch64.SettingAccessFlagPermitted(FaultRecord fault)
    if fault.statuscode == Fault_None then
        return TRUE;
    elsif fault.statuscode IN {Fault_Alignment, Fault_Permission} then
        return ConstrainUnpredictableBool(Unpredictable_AFUPDATE);
    else
        return FALSE;
```

Library pseudocode for aarch64/translation/vmsa_translation/AArch64.SettingDirtyStatePermitted

```
// AArch64.SettingDirtyStatePermitted()
// ====================================
// Determine whether the dirty state could be set by HW given the fault status

boolean AArch64.SettingDirtyStatePermitted(FaultRecord fault)
    if fault.statuscode == Fault_None then
        return TRUE;
    elsif fault.statuscode == Fault_Alignment then
        return ConstrainUnpredictableBool(Unpredictable_DBUPDATE);
    else
        return FALSE;
```

Library pseudocode for aarch64/translation/vmsa_translation/AArch64.TranslateAddress

```
// AArch64.TranslateAddress()
// ==========================
// Main entry point for translating an address

AddressDescriptor AArch64.TranslateAddress(bits(64) va, AccessDescriptor accdesc,
                                           boolean aligned, integer size)
    if (SPESampleInFlight && !(accdesc.acctype IN {AccessType_IFETCH,
                                                   AccessType_SPE})) then
        SPEStartCounter(SPECounterPosTranslationLatency);

    AddressDescriptor result = AArch64.FullTranslate(va, accdesc, aligned);

    if !IsFault(result) && accdesc.acctype != AccessType_IFETCH then
        result.fault = AArch64.CheckDebug(va, accdesc, size);

    if HaveRME() && !IsFault(result) && (
            accdesc.acctype != AccessType_DC ||
            boolean IMPLEMENTATION_DEFINED "GPC Fault on DC operations") then
        result.fault.gpcf = GranuleProtectionCheck(result, accdesc);

        if result.fault.gpcf.gpf != GPCF_None then
            result.fault.statuscode = Fault_GPCFOnOutput;
            result.fault.paddress   = result.paddress;

    if !IsFault(result) && accdesc.acctype == AccessType_IFETCH then
        result.fault = AArch64.CheckDebug(va, accdesc, size);

    if (SPESampleInFlight && !(accdesc.acctype IN {AccessType_IFETCH,
                                                   AccessType_SPE})) then
        SPEStopCounter(SPECounterPosTranslationLatency);

    // Update virtual address for abort functions
    result.vaddress = ZeroExtend(va, 64);

    return result;
```

Library pseudocode for aarch64/translation/vmsa_ttentry/AArch64.BlockDescSupported

```
// AArch64.BlockDescSupported()
// ============================
// Determine whether a block descriptor is valid for the given granule size
// and level

boolean AArch64.BlockDescSupported(bit d128, bit ds, TGx tgx, integer level)
    case tgx of
        when TGx_4KB  return ((level == 0 && (ds == '1' || d128 == '1')) ||
                              level == 1 ||
                              level == 2);
        when TGx_16KB return ((level == 1 && (ds == '1' || d128 == '1')) ||
                              level == 2);
        when TGx_64KB return ((level == 1 && (d128 == '1' || AArch64.PAMax() >= 52)) ||
                              level == 2);
    return FALSE;
```

Library pseudocode for aarch64/translation/vmsa_ttentry/AArch64.BlocknTFaults

```
// AArch64.BlocknTFaults()
// =======================
// Identify whether the nT bit in a block descriptor is effectively set
// causing a translation fault

boolean AArch64.BlocknTFaults(bit d128, bits(N) descriptor)
    bit nT;
    if !HaveBlockBBM() then
        return FALSE;
    nT = if d128 == '1' then descriptor<6> else descriptor<16>;
    bbm_level = AArch64.BlockBBMSupportLevel();
    nT_faults = (boolean IMPLEMENTATION_DEFINED
                 "BBM level 1 or 2 support nT bit causes Translation Fault");

    return bbm_level IN {1, 2} && nT == '1' && nT_faults;
```

Library pseudocode for aarch64/translation/vmsa_ttentry/AArch64.ContiguousBit

```
// AArch64.ContiguousBit()
// =======================
// Get the value of the contiguous bit

bit AArch64.ContiguousBit(TGx tgx, bit d128, integer level, bits(N) descriptor)
    if d128 == '1' then
        if (tgx == TGx_64KB && level == 1) || (tgx == TGx_4KB && level == 0) then
            return '0'; // RES0
        else
            return descriptor<111>;
    // When using TGx 64KB and FEAT_LPA is implememted,
    // the Contiguous bit is RES0 for Block descriptors at level 1

    if tgx == TGx_64KB && level == 1 then
        return '0'; // RES0

    // When the effective value of TCR_ELx.DS is '1',
    // the Contiguous bit is RES0 for all the following:
    //      * For TGx 4KB, Block descriptors at level 0
    //      * For TGx 16KB, Block descriptors at level 1

    if tgx == TGx_16KB && level == 1 then
        return '0'; // RES0

    if tgx == TGx_4KB  && level == 0 then
        return '0'; // RES0

    return descriptor<52>;
```

Library pseudocode for aarch64/translation/vmsa_ttentry/AArch64.DecodeDescriptorType

```
// AArch64.DecodeDescriptorType()
// ==============================
// Determine whether the descriptor is a page, block or table

DescriptorType AArch64.DecodeDescriptorType(bits(N) descriptor, bit d128, bit ds,
                                            TGx tgx, integer level)
    if descriptor<0> == '0' then
        return DescriptorType_Invalid;
    elsif d128 == '1' then
        bits(2) skl = descriptor<110:109>;
        if tgx IN {TGx_16KB, TGx_64KB} && UInt(skl) == 3 then
            return DescriptorType_Invalid;

        integer effective_level = level + UInt(skl);
        if effective_level > FINAL_LEVEL then
            return DescriptorType_Invalid;
        elsif effective_level == FINAL_LEVEL then
            return DescriptorType_Leaf;
        else
            return DescriptorType_Table;
    else
        if descriptor<1> == '1' then
            if level == FINAL_LEVEL then
                return DescriptorType_Leaf;
            else
                return DescriptorType_Table;
        elsif descriptor<1> == '0' then
            if AArch64.BlockDescSupported(d128, ds, tgx, level) then
                return DescriptorType_Leaf;
            else
                return DescriptorType_Invalid;
```

Library pseudocode for aarch64/translation/vmsa_ttentry/AArch64.S1ApplyOutputPerms

```
// AArch64.S1ApplyOutputPerms()
// ============================
// Apply output permissions encoded in stage 1 page/block descriptors

Permissions AArch64.S1ApplyOutputPerms(Permissions permissions_in, bits(N) descriptor,
                                       Regime regime, S1TTWParams walkparams)
    Permissions permissions = permissions_in;

    bits (4) pi_index;
    if walkparams.pie == '1' then
        if walkparams.d128 == '1' then
            pi_index = descriptor<118:115>;
        else
            pi_index = descriptor<54:53>:descriptor<51>:descriptor<6>;
        bit_index          = 4 * UInt(pi_index);
        permissions.ppi    = walkparams.pir<bit_index+3:bit_index>;
        permissions.upi    = walkparams.pire0<bit_index+3:bit_index>;
        permissions.ndirty = descriptor<7>;
    else
        if regime == Regime_EL10 && EL2Enabled() && walkparams.nv1 == '1' then
            permissions.ap<2:1> = descriptor<7>:'0';
            permissions.pxn     = descriptor<54>;
        elsif HasUnprivileged(regime) then
            permissions.ap<2:1> = descriptor<7:6>;
            permissions.uxn     = descriptor<54>;
            permissions.pxn     = descriptor<53>;
        else
            permissions.ap<2:1> = descriptor<7>:'1';
            permissions.xn      = descriptor<54>;
        // Descriptors marked with DBM set have the effective value of AP[2] cleared.
        // This implies no Permission faults caused by lack of write permissions are
        // reported, and the Dirty bit can be set.
        if walkparams.ha == '1' && walkparams.hd == '1' && descriptor<51> == '1' then
            permissions.ap<2> = '0';

    boolean poe   = AArch64.S1POEnabled(regime);
    boolean e0poe = HasUnprivileged(regime) && AArch64.S1E0POEnabled(regime, walkparams.nv1);

    if poe || e0poe then
        if walkparams.d128 == '1' then
            permissions.po_index = descriptor<124:121>;
        else
            permissions.po_index = '0':descriptor<62:60>;

    return permissions;
```

Library pseudocode for aarch64/translation/vmsa_ttentry/AArch64.S1ApplyTablePerms

```
// AArch64.S1ApplyTablePerms()
// ===========================
// Apply hierarchical permissions encoded in stage 1 table descriptors

Permissions AArch64.S1ApplyTablePerms(Permissions permissions_in, bits(N) descriptor,
                                      Regime regime, S1TTWParams walkparams)
    Permissions permissions = permissions_in;
    bits(2) ap_table;
    bit pxn_table;
    bit uxn_table;
    bit xn_table;
    if regime == Regime_EL10 && EL2Enabled() && walkparams.nv1 == '1' then
        if walkparams.d128 == '1' then
            ap_table  = descriptor<126>:'0';
            pxn_table = descriptor<124>;
        else
            ap_table  = descriptor<62>:'0';
            pxn_table = descriptor<60>;
        permissions.ap_table  = permissions.ap_table  OR ap_table;
        permissions.pxn_table = permissions.pxn_table OR pxn_table;

    elsif HasUnprivileged(regime) then
        if walkparams.d128 == '1' then
            ap_table  = descriptor<126:125>;
            uxn_table = descriptor<124>;
            pxn_table = descriptor<123>;
        else
            ap_table  = descriptor<62:61>;
            uxn_table = descriptor<60>;
            pxn_table = descriptor<59>;
        permissions.ap_table  = permissions.ap_table  OR ap_table;
        permissions.uxn_table = permissions.uxn_table OR uxn_table;
        permissions.pxn_table = permissions.pxn_table OR pxn_table;
    else
        if walkparams.d128 == '1' then
            ap_table = descriptor<126>:'0';
            xn_table = descriptor<124>;
        else
            ap_table = descriptor<62>:'0';
            xn_table = descriptor<60>;
        permissions.ap_table = permissions.ap_table OR ap_table;
        permissions.xn_table = permissions.xn_table OR xn_table;

    return permissions;
```

Library pseudocode for aarch64/translation/vmsa_ttentry/AArch64.S2ApplyOutputPerms

```
// AArch64.S2ApplyOutputPerms()
// ============================
// Apply output permissions encoded in stage 2 page/block descriptors

Permissions AArch64.S2ApplyOutputPerms(bits(N) descriptor, S2TTWParams walkparams)
    Permissions permissions;
    bits(4) s2pi_index;
    if walkparams.s2pie == '1' then
        if walkparams.d128 == '1' then
            s2pi_index = descriptor<118:115>;
        else
            s2pi_index = descriptor<54:53,51,6>;
        bit_index = 4 * UInt(s2pi_index);
        permissions.s2pi = walkparams.s2pir<bit_index+3 : bit_index>;
        permissions.s2dirty = descriptor<7>;
    else
        permissions.s2ap = descriptor<7:6>;
        if walkparams.d128 == '1' then
            permissions.s2xn = descriptor<118>;
        else
            permissions.s2xn = descriptor<54>;

        if HaveExtendedExecuteNeverExt() then
            if walkparams.d128 == '1' then
                permissions.s2xnx = descriptor<117>;
            else
                permissions.s2xnx = descriptor<53>;
        else
            permissions.s2xnx = '0';

        // Descriptors marked with DBM set have the effective value of S2AP[1] set.
        // This implies no Permission faults caused by lack of write permissions are
        // reported, and the Dirty bit can be set.
        bit desc_dbm;
        if walkparams.d128 == '1' then
            desc_dbm = descriptor<115>;
        else
            desc_dbm = descriptor<51>;
        if walkparams.ha == '1' && walkparams.hd == '1' && desc_dbm == '1' then
            permissions.s2ap<1> = '1';
    if walkparams.s2pie == '1' && HaveS2POExt() && VTCR_EL2.S2POE == '1' then
        if walkparams.d128 == '1' then
            permissions.s2po_index = descriptor<124:121>;
        else
            permissions.s2po_index = descriptor<62:59>;
    return permissions;
```

Library pseudocode for aarch64/translation/vmsa_walk/AArch64.S1InitialTTWState

```
// AArch64.S1InitialTTWState()
// ===========================
// Set properties of first access to translation tables in stage 1

TTWState AArch64.S1InitialTTWState(S1TTWParams walkparams, bits(64) va, Regime regime,
                                    SecurityState ss)
    TTWState    walkstate;
    FullAddress tablebase;
    Permissions permissions;
    bits(128)   ttbr;

    ttbr        = AArch64.S1TTBR(regime, va);
    case ss of
        when SS_Secure    tablebase.paspace = PAS_Secure;
        when SS_NonSecure tablebase.paspace = PAS_NonSecure;
        when SS_Root      tablebase.paspace = PAS_Root;
        when SS_Realm     tablebase.paspace = PAS_Realm;

    tablebase.address = AArch64.S1TTBaseAddress(walkparams, regime, ttbr);

    permissions.ap_table = '00';
    if HasUnprivileged(regime) then
        permissions.uxn_table = '0';
        permissions.pxn_table = '0';
    else
        permissions.xn_table  = '0';

    walkstate.baseaddress = tablebase;
    walkstate.level       = AArch64.S1StartLevel(walkparams);
    walkstate.istable     = TRUE;
    // In regimes that support global and non-global translations, translation
    // table entries from lookup levels other than the final level of lookup
    // are treated as being non-global
    walkstate.nG          = if HasUnprivileged(regime) then '1' else '0';
    walkstate.memattrs    = WalkMemAttrs(walkparams.sh, walkparams.irgn, walkparams.orgn);
    walkstate.permissions = permissions;
    if (walkparams.d128 == '1' || walkparams.pnch == '1') then
        walkstate.s1assured = TRUE;
    else
        walkstate.s1assured = FALSE;
    walkstate.disch = walkparams.disch;

    return walkstate;
```

Library pseudocode for aarch64/translation/vmsa_walk/AArch64.S1NextWalkStateLeaf

```
// AArch64.S1NextWalkStateLeaf()
// =============================
// Decode stage 1 page or block descriptor as output to this stage of translation

TTWState AArch64.S1NextWalkStateLeaf(TTWState currentstate, boolean s2fs1mro, Regime regime,
                                     SecurityState ss, S1TTWParams walkparams, bits(N) descriptor)
    TTWState    nextstate;
    FullAddress baseaddress;
    baseaddress.address = AArch64.LeafBase(descriptor, walkparams.d128,
                                               walkparams.ds,
                                               walkparams.tgx, currentstate.level);

    if currentstate.baseaddress.paspace == PAS_Secure then
        // Determine PA space of the block from NS bit
        bit ns;
        ns = if walkparams.d128 == '1' then descriptor<127> else descriptor<5>;
        baseaddress.paspace = if ns == '0' then PAS_Secure else PAS_NonSecure;
    elsif currentstate.baseaddress.paspace == PAS_Root then
        // Determine PA space of the block from NSE and NS bits
        bit nse;
        bit ns;
        <nse,ns> = if walkparams.d128 == '1' then descriptor<11,127> else descriptor<11,5>;
        baseaddress.paspace = DecodePASpace(nse, ns);

        // If Secure state is not implemented, but RME is,
        // force Secure space accesses to Non-secure space
        if baseaddress.paspace == PAS_Secure && !HaveSecureState() then
            baseaddress.paspace = PAS_NonSecure;

    elsif (currentstate.baseaddress.paspace == PAS_Realm &&
            regime IN {Regime_EL2, Regime_EL20}) then
        // Realm EL2 and EL2&0 regimes have a stage 1 NS bit
        bit ns;
        ns = if walkparams.d128 == '1' then descriptor<127> else descriptor<5>;
        baseaddress.paspace = if ns == '0' then PAS_Realm else PAS_NonSecure;
    elsif currentstate.baseaddress.paspace == PAS_Realm then
        // Realm EL1&0 regime does not have a stage 1 NS bit
        baseaddress.paspace = PAS_Realm;
    else
        baseaddress.paspace = PAS_NonSecure;

    nextstate.istable     = FALSE;
    nextstate.level       = currentstate.level;
    nextstate.baseaddress = baseaddress;

    bits(4) attrindx;
    if walkparams.aie == '1' then
        if walkparams.d128 == '1' then
            attrindx = descriptor<5:2>;
        else
            attrindx = descriptor<59,4:2>;
    else
        attrindx = '0':descriptor<4:2>;

    bits(2) sh;
    if walkparams.d128 == '1' then
        sh = descriptor<9:8>;
    elsif walkparams.ds == '1' then
        sh = walkparams.sh;
    else
        sh = descriptor<9:8>;
    attr = AArch64.MAIRAttr(UInt(attrindx), walkparams.mair2, walkparams.mair);
    s1aarch64 = TRUE;

    nextstate.memattrs    = S1DecodeMemAttrs(attr, sh, s1aarch64, walkparams);
    nextstate.permissions = AArch64.S1ApplyOutputPerms(currentstate.permissions,
                                                           descriptor, regime, walkparams);
    bit protectedbit;
    if walkparams.d128 == '1' then
        protectedbit = descriptor<114>;
```

```
    else
        protectedbit = if walkparams.pnch == '1' then descriptor<52> else '0';
    if (currentstate.s1assured && s2fs1mro && protectedbit == '1') then
        nextstate.s1assured = TRUE;
    else
        nextstate.s1assured = FALSE;

    if walkparams.pnch == '1' || currentstate.disch == '1' then
        nextstate.contiguous = '0';
    else
        nextstate.contiguous = AArch64.ContiguousBit(walkparams.tgx, walkparams.d128,
                                                     currentstate.level, descriptor);
    if !HasUnprivileged(regime) then
        nextstate.nG = '0';
    elsif ss == SS_Secure && currentstate.baseaddress.paspace == PAS_NonSecure then
        // In Secure state, a translation must be treated as non-global,
        // regardless of the value of the nG bit,
        // if NSTable is set to 1 at any level of the translation table walk
        nextstate.nG = '1';
    else
        nextstate.nG = descriptor<11>;

    if walkparams.d128 == '1' then
        nextstate.guardedpage = descriptor<113>;
    else
        nextstate.guardedpage = descriptor<50>;

    return nextstate;
```

Library pseudocode for aarch64/translation/vmsa_walk/AArch64.S1NextWalkStateTable

```
// AArch64.S1NextWalkStateTable()
// ==============================
// Decode stage 1 table descriptor to transition to the next level

TTWState AArch64.S1NextWalkStateTable(TTWState currentstate, boolean s2fs1mro, Regime regime,
                                      S1TTWParams walkparams, bits(N) descriptor)
    TTWState    nextstate;
    FullAddress tablebase;

    tablebase.address = AArch64.NextTableBase(descriptor, walkparams.d128,
                                              walkparams.ds,
                                              walkparams.tgx);
    if currentstate.baseaddress.paspace == PAS_Secure then
        // Determine PA space of the next table from NSTable bit
        bit nstable;
        nstable = if walkparams.d128 == '1' then descriptor<127> else descriptor<63>;
        tablebase.paspace = if nstable == '0' then PAS_Secure else PAS_NonSecure;
    else
        // Otherwise bit 63 is RES0 and there is no NSTable bit
        tablebase.paspace = currentstate.baseaddress.paspace;

    nextstate.istable     = TRUE;
    nextstate.nG          = currentstate.nG;
    if walkparams.d128 == '1' then
        skl = descriptor<110:109>;
        nextstate.level   = currentstate.level + UInt(skl) + 1;
    else
        nextstate.level   = currentstate.level + 1;
    nextstate.baseaddress = tablebase;
    nextstate.memattrs    = currentstate.memattrs;
    if walkparams.hpd == '0' && walkparams.pie == '0' then
        nextstate.permissions = AArch64.S1ApplyTablePerms(currentstate.permissions, descriptor,
                                                          regime, walkparams);
    else
        nextstate.permissions = currentstate.permissions;
    bit protectedbit;
    if walkparams.d128 == '1' then
        protectedbit = descriptor<114>;
    else
        protectedbit = if walkparams.pnch == '1' then descriptor<52> else '0';
    if (currentstate.s1assured && s2fs1mro && protectedbit == '1') then
        nextstate.s1assured = TRUE;
    else
        nextstate.s1assured = FALSE;
    nextstate.disch = if walkparams.d128 == '1' then descriptor<112> else '0';

    return nextstate;
```

Library pseudocode for aarch64/translation/vmsa_walk/AArch64.S1Walk

```
// AArch64.S1Walk()
// ================
// Traverse stage 1 translation tables obtaining the final descriptor
// as well as the address leading to that descriptor

(FaultRecord, AddressDescriptor, TTWState, bits(N)) AArch64.S1Walk(FaultRecord fault_in,
                                                                   S1TTWParams walkparams,
                                                                   bits(64) va, Regime regime,
                                                                   AccessDescriptor accdesc,
                                                                   integer N)
    FaultRecord fault = fault_in;
    boolean s1aarch64;
    boolean aligned;

    if HasUnprivileged(regime) && AArch64.S1EPD(regime, va) == '1' then
        fault.statuscode = Fault_Translation;
        fault.level      = 0;
        return (fault, AddressDescriptor UNKNOWN, TTWState UNKNOWN,
                bits(N) UNKNOWN);

    walkstate = AArch64.S1InitialTTWState(walkparams, va, regime, accdesc.ss);
    constant integer startlevel = walkstate.level;

    bits(N) descriptor;
    AddressDescriptor walkaddress;
    bits(2) skl = '00';
    walkaddress.vaddress = va;
    walkaddress.mecid = AArch64.TTWalkMECID(walkparams.emec, regime, accdesc.ss);

    if !AArch64.S1DCacheEnabled(regime) then
        walkaddress.memattrs = NormalNCMemAttr();
        walkaddress.memattrs.xs = walkstate.memattrs.xs;
    else
        walkaddress.memattrs = walkstate.memattrs;

    // Shareability value of stage 1 translation subject to stage 2 is IMPLEMENTATION DEFINED
    // to be either effective value or descriptor value
    if (regime == Regime_EL10 && EL2Enabled() && HCR_EL2.VM == '1' &&
            !(boolean IMPLEMENTATION_DEFINED "Apply effective shareability at stage 1")) then
        walkaddress.memattrs.shareability = walkstate.memattrs.shareability;
    else
        walkaddress.memattrs.shareability = EffectiveShareability(walkaddress.memattrs);

    boolean s2fs1mro = FALSE;

    DescriptorType desctype;
    FullAddress descaddress = AArch64.S1SLTTEntryAddress(walkstate.level, walkparams, va,
                                                         walkstate.baseaddress);

    // Detect Address Size Fault by Descriptor Address
    if AArch64.OAOutOfRange(descaddress.address, walkparams.d128,
                            walkparams.ps, walkparams.tgx) then
        fault.statuscode = Fault_AddressSize;
        fault.level      = 0;
        return (fault, AddressDescriptor UNKNOWN, TTWState UNKNOWN, bits(N) UNKNOWN);

    repeat
        fault.level = walkstate.level;
        walkaddress.paddress = descaddress;

        boolean toplevel = walkstate.level == startlevel;
        VARange varange  = AArch64.GetVARange(va);
        AccessDescriptor walkaccess = CreateAccDescS1TTW(toplevel, varange, accdesc);
        FaultRecord s2fault;
        AddressDescriptor s2walkaddress;
        if regime == Regime_EL10 && EL2Enabled() then
            s1aarch64 = TRUE;
            aligned   = TRUE;
            (s2fault, s2walkaddress) = AArch64.S2Translate(fault, walkaddress, s1aarch64, aligned,
                                                           walkaccess);
```

```
        if s2fault.statuscode != Fault_None then
            return (s2fault, AddressDescriptor UNKNOWN, TTWState UNKNOWN,
                    bits(N) UNKNOWN);

        s2fs1mro = s2walkaddress.s2fs1mro;
        (fault, descriptor) = FetchDescriptor(walkparams.ee, s2walkaddress, walkaccess,
                                              fault, N);
    else
        (fault, descriptor) = FetchDescriptor(walkparams.ee, walkaddress, walkaccess,
                                              fault, N);

    if fault.statuscode != Fault_None then
        return (fault, AddressDescriptor UNKNOWN, TTWState UNKNOWN,
                bits(N) UNKNOWN);

    bits(N) new_descriptor;
    repeat
        new_descriptor = descriptor;
        desctype = AArch64.DecodeDescriptorType(descriptor, walkparams.d128, walkparams.ds,
                                                walkparams.tgx, walkstate.level);
        case desctype of
            when DescriptorType_Table
                walkstate = AArch64.S1NextWalkStateTable(walkstate, s2fs1mro,
                                                         regime, walkparams, descriptor);
                skl = if walkparams.d128 == '1' then descriptor<110:109> else '00';
                descaddress = AArch64.TTEntryAddress(walkstate.level, walkparams.d128, skl,
                                                     walkparams.tgx, walkparams.txsz, va,
                                                     walkstate.baseaddress);

                // Detect Address Size Fault by Descriptor Address
                if AArch64.OAOutOfRange(descaddress.address, walkparams.d128,
                                        walkparams.ps, walkparams.tgx) then
                    fault.statuscode = Fault_AddressSize;
                    return (fault, AddressDescriptor UNKNOWN, TTWState UNKNOWN,
                            bits(N) UNKNOWN);

                if walkparams.haft == '1' then
                    new_descriptor<10> = '1';
                if (walkparams.d128 == '1' && skl != '00' &&
                      AArch64.BlocknTFaults(walkparams.d128, descriptor)) then
                    fault.statuscode = Fault_Translation;
                    return (fault, AddressDescriptor UNKNOWN, TTWState UNKNOWN,
                            bits(N) UNKNOWN);
            when DescriptorType_Leaf
                walkstate = AArch64.S1NextWalkStateLeaf(walkstate, s2fs1mro,
                                                        regime, accdesc.ss, walkparams,
                                                        descriptor);
            when DescriptorType_Invalid
                fault.statuscode = Fault_Translation;
                return (fault, AddressDescriptor UNKNOWN, TTWState UNKNOWN,
                        bits(N) UNKNOWN);
            otherwise
                Unreachable();

        if new_descriptor != descriptor then
            AddressDescriptor descpaddr;
            AccessDescriptor descaccess = CreateAccDescTTEUpdate(accdesc);
            if regime == Regime_EL10 && EL2Enabled() then
                s1aarch64 = TRUE;
                aligned   = TRUE;
                (s2fault, descpaddr) = AArch64.S2Translate(fault, walkaddress,
                                                           s1aarch64, aligned,
                                                           descaccess);

                if s2fault.statuscode != Fault_None then
                    return (s2fault, AddressDescriptor UNKNOWN,
                            TTWState UNKNOWN, bits(N) UNKNOWN);
            else
                descpaddr = walkaddress;
```

```
                (fault, descriptor) = AArch64.MemSwapTableDesc(fault, descriptor, new_descriptor,
                                                         walkparams.ee, descaccess,
                                                         descpaddr);
                if fault.statuscode != Fault_None then
                    return (fault, AddressDescriptor UNKNOWN,
                            TTWState UNKNOWN, bits(N) UNKNOWN);
        until new_descriptor == descriptor;
    until desctype == DescriptorType_Leaf;

    FullAddress oa = StageOA(va, walkparams.d128, walkparams.tgx, walkstate);

    if (walkstate.contiguous == '1' &&
        AArch64.ContiguousBitFaults(walkparams.d128, walkparams.txsz, walkparams.tgx,
                                    walkstate.level)) then
        fault.statuscode = Fault_Translation;
    elsif (desctype == DescriptorType_Leaf && walkstate.level < FINAL_LEVEL &&
             AArch64.BlocknTFaults(walkparams.d128, descriptor)) then
        fault.statuscode = Fault_Translation;
    elsif AArch64.S1AMECFault(walkparams, walkstate.baseaddress.paspace, regime, descriptor) then
        fault.statuscode = Fault_Translation;
    // Detect Address Size Fault by final output
    elsif AArch64.OAOutOfRange(oa.address, walkparams.d128,
                               walkparams.ps, walkparams.tgx) then
        fault.statuscode = Fault_AddressSize;
    // Check descriptor AF bit
    elsif (descriptor<10> == '0' && walkparams.ha == '0' &&
            !(accdesc.acctype IN {AccessType_DC, AccessType_IC} &&
            !boolean IMPLEMENTATION_DEFINED "Generate access flag fault on IC/DC operations")) then
        fault.statuscode = Fault_AccessFlag;

    if fault.statuscode != Fault_None then
        return (fault, AddressDescriptor UNKNOWN, TTWState UNKNOWN, bits(N) UNKNOWN);

    return (fault, walkaddress, walkstate, descriptor);
```

Library pseudocode for aarch64/translation/vmsa_walk/AArch64.S2InitialTTWState

```
// AArch64.S2InitialTTWState()
// ===========================
// Set properties of first access to translation tables in stage 2

TTWState AArch64.S2InitialTTWState(SecurityState ss, S2TTWParams walkparams)
    TTWState    walkstate;
    FullAddress tablebase;
    bits(128)   ttbr;

    ttbr = ZeroExtend(VTTBR_EL2, 128);
    case ss of
        when SS_NonSecure tablebase.paspace = PAS_NonSecure;
        when SS_Realm     tablebase.paspace = PAS_Realm;
    tablebase.address = AArch64.S2TTBaseAddress(walkparams, tablebase.paspace, ttbr);

    walkstate.baseaddress = tablebase;
    walkstate.level       = AArch64.S2StartLevel(walkparams);
    walkstate.istable     = TRUE;
    walkstate.memattrs    = WalkMemAttrs(walkparams.sh, walkparams.irgn, walkparams.orgn);

    return walkstate;
```

Library pseudocode for aarch64/translation/vmsa_walk/AArch64.S2NextWalkStateLeaf

```
// AArch64.S2NextWalkStateLeaf()
// =============================
// Decode stage 2 page or block descriptor as output to this stage of translation

TTWState AArch64.S2NextWalkStateLeaf(TTWState currentstate, SecurityState ss,
                                     S2TTWParams walkparams, AddressDescriptor ipa,
                                     bits(N) descriptor)
    TTWState    nextstate;
    FullAddress baseaddress;

    if ss == SS_Secure then
        baseaddress.paspace = AArch64.SS2OutputPASpace(walkparams, ipa.paddress.paspace);
    elsif ss == SS_Realm then
        bit ns;
        ns = if walkparams.d128 == '1' then descriptor<127> else descriptor<55>;
        baseaddress.paspace = if ns == '1' then PAS_NonSecure else PAS_Realm;
    else
        baseaddress.paspace = PAS_NonSecure;
    baseaddress.address   = AArch64.LeafBase(descriptor, walkparams.d128, walkparams.ds,
                                             walkparams.tgx, currentstate.level);

    nextstate.istable     = FALSE;
    nextstate.level       = currentstate.level;
    nextstate.baseaddress = baseaddress;
    nextstate.permissions = AArch64.S2ApplyOutputPerms(descriptor, walkparams);

    s2_attr = descriptor<5:2>;
    s2_sh   = if walkparams.ds == '1' then walkparams.sh else descriptor<9:8>;
    s2_fnxs = descriptor<11>;
    if walkparams.fwb == '1' then
        nextstate.memattrs = AArch64.S2ApplyFWBMemAttrs(ipa.memattrs, walkparams, descriptor);
        if s2_attr<3:1> == '111' then
            nextstate.permissions.s2tag_na = '1';
        else
            nextstate.permissions.s2tag_na = '0';
    else
        s2aarch64 = TRUE;
        nextstate.memattrs = S2DecodeMemAttrs(s2_attr, s2_sh, s2aarch64);
        // FnXS is used later to mask the XS value from stage 1
        nextstate.memattrs.xs = NOT s2_fnxs;
        if s2_attr == '0100' then
            nextstate.permissions.s2tag_na = '1';
        else
            nextstate.permissions.s2tag_na = '0';
    nextstate.contiguous = AArch64.ContiguousBit(walkparams.tgx, walkparams.d128,
                                                 currentstate.level, descriptor);
    if walkparams.d128 == '1' then
        nextstate.s2assuredonly = descriptor<114>;
    else
        nextstate.s2assuredonly = if walkparams.assuredonly == '1' then descriptor<58> else '0';

    return nextstate;
```

Library pseudocode for aarch64/translation/vmsa_walk/AArch64.S2NextWalkStateTable

```
// AArch64.S2NextWalkStateTable()
// ==============================
// Decode stage 2 table descriptor to transition to the next level

TTWState AArch64.S2NextWalkStateTable(TTWState currentstate, S2TTWParams walkparams,
                                      bits(N) descriptor)
    TTWState    nextstate;
    FullAddress tablebase;

    tablebase.address = AArch64.NextTableBase(descriptor, walkparams.d128,
                                              walkparams.ds,
                                              walkparams.tgx);
    tablebase.paspace = currentstate.baseaddress.paspace;

    nextstate.istable     = TRUE;
    if walkparams.d128 == '1' then
        skl = descriptor<110:109>;
        nextstate.level   = currentstate.level + UInt(skl) + 1;
    else
        nextstate.level   = currentstate.level + 1;
    nextstate.baseaddress = tablebase;
    nextstate.memattrs    = currentstate.memattrs;

    return nextstate;
```

Library pseudocode for aarch64/translation/vmsa_walk/AArch64.S2Walk

```
// AArch64.S2Walk()
// ================
// Traverse stage 2 translation tables obtaining the final descriptor
// as well as the address leading to that descriptor

(FaultRecord, AddressDescriptor, TTWState, bits(N)) AArch64.S2Walk(FaultRecord fault_in,
                                                                   AddressDescriptor ipa,
                                                                   S2TTWParams walkparams,
                                                                   AccessDescriptor accdesc,
                                                                   integer N)

    FaultRecord fault = fault_in;
    ipa_64 = ZeroExtend(ipa.paddress.address, 64);

    TTWState walkstate;
    if accdesc.ss == SS_Secure then
        walkstate = AArch64.SS2InitialTTWState(walkparams, ipa.paddress.paspace);
    else
        walkstate = AArch64.S2InitialTTWState(accdesc.ss, walkparams);

    constant integer startlevel = walkstate.level;

    bits(N) descriptor;
    AccessDescriptor walkaccess = CreateAccDescS2TTW(accdesc);
    AddressDescriptor walkaddress;
    bits(2) skl = '00';

    walkaddress.vaddress = ipa.vaddress;
    walkaddress.mecid = AArch64.TTWalkMECID(walkparams.emec, Regime_EL10, accdesc.ss);

    if HCR_EL2.CD == '1' then
        walkaddress.memattrs = NormalNCMemAttr();
        walkaddress.memattrs.xs = walkstate.memattrs.xs;
    else
        walkaddress.memattrs = walkstate.memattrs;

    walkaddress.memattrs.shareability = EffectiveShareability(walkaddress.memattrs);

    DescriptorType desctype;

    // Initial lookup might index into concatenated tables
    FullAddress descaddress = AArch64.S2SLTTEntryAddress(walkparams, ipa.paddress.address,
                                                         walkstate.baseaddress);

    // Detect Address Size Fault by Descriptor Address
    if AArch64.OAOutOfRange(descaddress.address, walkparams.d128, walkparams.ps,
                            walkparams.tgx) then
        fault.statuscode = Fault_AddressSize;
        fault.level      = 0;
        return (fault, AddressDescriptor UNKNOWN, TTWState UNKNOWN, bits(N) UNKNOWN);

    repeat
        fault.level = walkstate.level;
        walkaddress.paddress = descaddress;
        (fault, descriptor) = FetchDescriptor(walkparams.ee, walkaddress, walkaccess, fault, N);

        if fault.statuscode != Fault_None then
            return (fault, AddressDescriptor UNKNOWN, TTWState UNKNOWN, bits(N) UNKNOWN);

        bits(N) new_descriptor;
        repeat
            new_descriptor = descriptor;
            desctype = AArch64.DecodeDescriptorType(descriptor, walkparams.d128, walkparams.ds,
                                                    walkparams.tgx, walkstate.level);
            case desctype of
                when DescriptorType_Table
                    walkstate = AArch64.S2NextWalkStateTable(walkstate, walkparams, descriptor);
                    skl = if walkparams.d128 == '1' then descriptor<110:109> else '00';
                    descaddress = AArch64.TTEntryAddress(walkstate.level, walkparams.d128, skl,
                                                         walkparams.tgx, walkparams.txsz, ipa_64,
```

```
                                                      walkstate.baseaddress);

                    // Detect Address Size Fault by table descriptor
                    if AArch64.OAOutOfRange(descaddress.address, walkparams.d128, walkparams.ps,
                                             walkparams.tgx) then
                        fault.statuscode = Fault_AddressSize;
                        return (fault, AddressDescriptor UNKNOWN, TTWState UNKNOWN,
                                bits(N) UNKNOWN);

                    if walkparams.haft == '1' then
                        new_descriptor<10> = '1';

                    if (walkparams.d128 == '1' && skl != '00' &&
                          AArch64.BlocknTFaults(walkparams.d128, descriptor)) then
                        fault.statuscode = Fault_Translation;
                        return (fault, AddressDescriptor UNKNOWN, TTWState UNKNOWN,
                                bits(N) UNKNOWN);

                when DescriptorType_Leaf
                    walkstate = AArch64.S2NextWalkStateLeaf(walkstate, accdesc.ss, walkparams, ipa,
                                                            descriptor);
                when DescriptorType_Invalid
                    fault.statuscode = Fault_Translation;
                    return (fault, AddressDescriptor UNKNOWN, TTWState UNKNOWN, bits(N) UNKNOWN);

                otherwise
                    Unreachable();

            if new_descriptor != descriptor then
                AccessDescriptor descaccess = CreateAccDescTTEUpdate(accdesc);
                (fault, descriptor) = AArch64.MemSwapTableDesc(fault, descriptor, new_descriptor,
                                                               walkparams.ee, descaccess,
                                                               walkaddress);
                if fault.statuscode != Fault_None then
                    return (fault, AddressDescriptor UNKNOWN, TTWState UNKNOWN, bits(N) UNKNOWN);
        until new_descriptor == descriptor;
    until desctype == DescriptorType_Leaf;

    FullAddress oa = StageOA(ipa_64, walkparams.d128, walkparams.tgx, walkstate);

    if (walkstate.contiguous == '1' &&
        AArch64.ContiguousBitFaults(walkparams.d128, walkparams.txsz, walkparams.tgx,
                                    walkstate.level)) then
        fault.statuscode = Fault_Translation;
    elsif (desctype == DescriptorType_Leaf && walkstate.level < FINAL_LEVEL &&
             AArch64.BlocknTFaults(walkparams.d128, descriptor)) then
        fault.statuscode = Fault_Translation;
    // Detect Address Size Fault by final output
    elsif AArch64.OAOutOfRange(oa.address, walkparams.d128, walkparams.ps, walkparams.tgx) then
        fault.statuscode = Fault_AddressSize;
    // Check descriptor AF bit
    elsif (descriptor<10> == '0' && walkparams.ha == '0' &&
            !(accdesc.acctype IN {AccessType_DC, AccessType_IC} &&
            !boolean IMPLEMENTATION_DEFINED "Generate access flag fault on IC/DC operations")) then
        fault.statuscode = Fault_AccessFlag;

    return (fault, walkaddress, walkstate, descriptor);
```

Library pseudocode for aarch64/translation/vmsa_walk/AArch64.SS2InitialTTWState

```
// AArch64.SS2InitialTTWState()
// ============================
// Set properties of first access to translation tables in Secure stage 2

TTWState AArch64.SS2InitialTTWState(S2TTWParams walkparams, PASpace ipaspace)
    TTWState    walkstate;
    FullAddress tablebase;
    bits(128)   ttbr;

    if ipaspace == PAS_Secure then
        ttbr = ZeroExtend(VSTTBR_EL2, 128);
    else
        ttbr = ZeroExtend(VTTBR_EL2, 128);

    if ipaspace == PAS_Secure then
        if walkparams.sw == '0' then
            tablebase.paspace = PAS_Secure;
        else
            tablebase.paspace = PAS_NonSecure;
    else
        if walkparams.nsw == '0' then
            tablebase.paspace = PAS_Secure;
        else
            tablebase.paspace = PAS_NonSecure;

    tablebase.address = AArch64.S2TTBaseAddress(walkparams, tablebase.paspace, ttbr);

    walkstate.baseaddress = tablebase;
    walkstate.level       = AArch64.S2StartLevel(walkparams);
    walkstate.istable     = TRUE;
    walkstate.memattrs    = WalkMemAttrs(walkparams.sh, walkparams.irgn, walkparams.orgn);

    return walkstate;
```

Library pseudocode for aarch64/translation/vmsa_walk/AArch64.SS2OutputPASpace

```
// AArch64.SS2OutputPASpace()
// ==========================
// Assign PA Space to output of Secure stage 2 translation

PASpace AArch64.SS2OutputPASpace(S2TTWParams walkparams, PASpace ipaspace)
    if ipaspace == PAS_Secure then
        if walkparams.<sw,sa> == '00' then
            return PAS_Secure;
        else
            return PAS_NonSecure;
    else
        if walkparams.<sw,sa,nsw,nsa> == '0000' then
            return PAS_Secure;
        else
            return PAS_NonSecure;
```

Library pseudocode for aarch64/translation/vmsa_walkparams/AArch64.BBMSupportLevel

```
// AArch64.BBMSupportLevel()
// =========================
// Returns the level of FEAT_BBM supported

integer AArch64.BlockBBMSupportLevel()
    if !HaveBlockBBM() then
        return integer UNKNOWN;
    else
        return integer IMPLEMENTATION_DEFINED "Block BBM support level";
```

Library pseudocode for aarch64/translation/vmsa_walkparams/AArch64.GetS1TTWParams

```
// AArch64.GetS1TTWParams()
// ========================
// Returns stage 1 translation table walk parameters from respective controlling
// System registers.

S1TTWParams AArch64.GetS1TTWParams(Regime regime, SecurityState ss, bits(64) va)
    S1TTWParams walkparams;

    varange = AArch64.GetVARange(va);

    case regime of
        when Regime_EL3  walkparams = AArch64.S1TTWParamsEL3();
        when Regime_EL2  walkparams = AArch64.S1TTWParamsEL2(ss);
        when Regime_EL20 walkparams = AArch64.S1TTWParamsEL20(ss, varange);
        when Regime_EL10 walkparams = AArch64.S1TTWParamsEL10(varange);

    return walkparams;
```

Library pseudocode for aarch64/translation/vmsa_walkparams/AArch64.GetS2TTWParams

```
// AArch64.GetS2TTWParams()
// ========================
// Gather walk parameters for stage 2 translation

S2TTWParams AArch64.GetS2TTWParams(SecurityState ss, PASpace ipaspace, boolean s1aarch64)
    S2TTWParams walkparams;

    if ss == SS_NonSecure then
        walkparams = AArch64.NSS2TTWParams(s1aarch64);
    elsif HaveSecureEL2Ext() && ss == SS_Secure then
        walkparams = AArch64.SS2TTWParams(ipaspace, s1aarch64);
    elsif ss == SS_Realm then
        walkparams = AArch64.RLS2TTWParams(s1aarch64);
    else
        Unreachable();

    return walkparams;
```

Library pseudocode for aarch64/translation/vmsa_walkparams/AArch64.GetVARange

```
// AArch64.GetVARange()
// ====================
// Determines if the VA that is to be translated lies in LOWER or UPPER address range.

VARange AArch64.GetVARange(bits(64) va)
    if va<55> == '0' then
        return VARange_LOWER;
    else
        return VARange_UPPER;
```

Library pseudocode for aarch64/translation/vmsa_walkparams/AArch64.HaveS1TG

```
// AArch64.HaveS1TG()
// ==================
// Determine whether the given translation granule is supported for stage 1

boolean AArch64.HaveS1TG(TGx tgx)
    case tgx of
        when TGx_4KB  return boolean IMPLEMENTATION_DEFINED "Has 4K Translation Granule";
        when TGx_16KB return boolean IMPLEMENTATION_DEFINED "Has 16K Translation Granule";
        when TGx_64KB return boolean IMPLEMENTATION_DEFINED "Has 64K Translation Granule";
```

Library pseudocode for aarch64/translation/vmsa_walkparams/AArch64.HaveS2TG

```
// AArch64.HaveS2TG()
// ==================
// Determine whether the given translation granule is supported for stage 2

boolean AArch64.HaveS2TG(TGx tgx)
    assert HaveEL(EL2);

    if HaveGTGExt() then
        case tgx of
            when TGx_4KB
                return boolean IMPLEMENTATION_DEFINED "Has Stage 2 4K Translation Granule";
            when TGx_16KB
                return boolean IMPLEMENTATION_DEFINED "Has Stage 2 16K Translation Granule";
            when TGx_64KB
                return boolean IMPLEMENTATION_DEFINED "Has Stage 2 64K Translation Granule";
    else
        return AArch64.HaveS1TG(tgx);
```

Library pseudocode for aarch64/translation/vmsa_walkparams/AArch64.MaxTxSZ

```
// AArch64.MaxTxSZ()
// =================
// Retrieve the maximum value of TxSZ indicating minimum input address size for both
// stages of translation

integer AArch64.MaxTxSZ(TGx tgx)
    if HaveSmallTranslationTableExt() then
        case tgx of
            when TGx_4KB   return 48;
            when TGx_16KB  return 48;
            when TGx_64KB  return 47;

    return 39;
```

Library pseudocode for aarch64/translation/vmsa_walkparams/AArch64.NSS2TTWParams

```
// AArch64.NSS2TTWParams()
// =======================
// Gather walk parameters specific for Non-secure stage 2 translation

S2TTWParams AArch64.NSS2TTWParams(boolean s1aarch64)
    S2TTWParams walkparams;

    walkparams.vm   = HCR_EL2.VM OR HCR_EL2.DC;
    walkparams.tgx  = AArch64.S2DecodeTG0(VTCR_EL2.TG0);
    walkparams.txsz = VTCR_EL2.T0SZ;
    walkparams.ps   = VTCR_EL2.PS;
    walkparams.irgn = VTCR_EL2.IRGN0;
    walkparams.orgn = VTCR_EL2.ORGN0;
    walkparams.sh   = VTCR_EL2.SH0;
    walkparams.ee   = SCTLR_EL2.EE;
    walkparams.d128 = if Have128BitDescriptorExt() then VTCR_EL2.D128 else '0';
    if walkparams.d128 == '1' then
        walkparams.skl = VTTBR_EL2.SKL;
    else
        walkparams.sl0 = VTCR_EL2.SL0;

    walkparams.ptw = if HCR_EL2.TGE == '0'         then HCR_EL2.PTW else '0';
    walkparams.fwb = if HaveStage2MemAttrControl() then HCR_EL2.FWB else '0';
    walkparams.ha  = if HaveAccessFlagUpdateExt()  then VTCR_EL2.HA else '0';
    walkparams.hd  = if HaveDirtyBitModifierExt()  then VTCR_EL2.HD else '0';
    if walkparams.tgx IN {TGx_4KB, TGx_16KB} && Have52BitIPAAndPASpaceExt() then
        walkparams.ds = VTCR_EL2.DS;
    else
        walkparams.ds = '0';
    if walkparams.tgx == TGx_4KB && Have52BitIPAAndPASpaceExt() then
        walkparams.sl2 = VTCR_EL2.SL2 AND VTCR_EL2.DS;
    else
        walkparams.sl2 = '0';
    walkparams.cmow = if HaveFeatCMOW() && IsHCRXEL2Enabled() then HCRX_EL2.CMOW else '0';
    if walkparams.d128 == '1' then
        walkparams.s2pie = '1';
    else
        walkparams.s2pie = if HaveS2PIExt() then VTCR_EL2.S2PIE else '0';
    walkparams.s2pir = if HaveS2PIExt() then S2PIR_EL2 else Zeros(64);
    if HaveTHExt() && walkparams.d128 != '1' then
        walkparams.assuredonly = VTCR_EL2.AssuredOnly;
    else
        walkparams.assuredonly = '0';
    walkparams.tl0   = if HaveTHExt() then VTCR_EL2.TL0 else '0';
    walkparams.tl1   = if HaveTHExt() then VTCR_EL2.TL1 else '0';
    if HaveAccessFlagUpdateForTableExt() && walkparams.ha == '1' then
        walkparams.haft = VTCR_EL2.HAFT;
    else
        walkparams.haft = '0';

    return walkparams;
```

Library pseudocode for aarch64/translation/vmsa_walkparams/AArch64.PAMax

```
// AArch64.PAMax()
// ===============
// Returns the IMPLEMENTATION DEFINED maximum number of bits capable of representing
// physical address for this processor

integer AArch64.PAMax()
    return integer IMPLEMENTATION_DEFINED "Maximum Physical Address Size";
```

Library pseudocode for aarch64/translation/vmsa_walkparams/AArch64.RLS2TTWParams

```
// AArch64.RLS2TTWParams()
// =======================
// Gather walk parameters specific for Realm stage 2 translation

S2TTWParams AArch64.RLS2TTWParams(boolean s1aarch64)
    // Realm stage 2 walk parameters are similar to Non-secure
    S2TTWParams walkparams = AArch64.NSS2TTWParams(s1aarch64);
    walkparams.emec = if HaveFeatMEC() && IsSCTLR2EL2Enabled() then SCTLR2_EL2.EMEC else '0';
    return walkparams;
```

Library pseudocode for aarch64/translation/vmsa_walkparams/AArch64.S1DCacheEnabled

```
// AArch64.S1DCacheEnabled()
// =========================
// Determine cacheability of stage 1 data accesses

boolean AArch64.S1DCacheEnabled(Regime regime)
    case regime of
        when Regime_EL3  return SCTLR_EL3.C == '1';
        when Regime_EL2  return SCTLR_EL2.C == '1';
        when Regime_EL20 return SCTLR_EL2.C == '1';
        when Regime_EL10 return SCTLR_EL1.C == '1';
```

Library pseudocode for aarch64/translation/vmsa_walkparams/AArch64.S1DecodeTG0

```
// AArch64.S1DecodeTG0()
// =====================
// Decode stage 1 granule size configuration bits TG0

TGx AArch64.S1DecodeTG0(bits(2) tg0_in)
    bits(2) tg0 = tg0_in;
    TGx tgx;

    if tg0 == '11' then
        tg0 = bits(2) IMPLEMENTATION_DEFINED "TG0 encoded granule size";

    case tg0 of
        when '00'   tgx = TGx_4KB;
        when '01'   tgx = TGx_64KB;
        when '10'   tgx = TGx_16KB;

    if !AArch64.HaveS1TG(tgx) then
        case bits(2) IMPLEMENTATION_DEFINED "TG0 encoded granule size" of
            when '00'   tgx = TGx_4KB;
            when '01'   tgx = TGx_64KB;
            when '10'   tgx = TGx_16KB;

    return tgx;
```

Library pseudocode for aarch64/translation/vmsa_walkparams/AArch64.S1DecodeTG1

```
// AArch64.S1DecodeTG1()
// =====================
// Decode stage 1 granule size configuration bits TG1

TGx AArch64.S1DecodeTG1(bits(2) tg1_in)
    bits(2) tg1 = tg1_in;
    TGx tgx;

    if tg1 == '00' then
        tg1 = bits(2) IMPLEMENTATION_DEFINED "TG1 encoded granule size";

    case tg1 of
        when '10'   tgx = TGx_4KB;
        when '11'   tgx = TGx_64KB;
        when '01'   tgx = TGx_16KB;

    if !AArch64.HaveS1TG(tgx) then
        case bits(2) IMPLEMENTATION_DEFINED "TG1 encoded granule size" of
            when '10'   tgx = TGx_4KB;
            when '11'   tgx = TGx_64KB;
            when '01'   tgx = TGx_16KB;

    return tgx;
```

Library pseudocode for aarch64/translation/vmsa_walkparams/AArch64.S1E0POEnabled

```
// AArch64.S1E0POEnabled()
// =======================
// Determine whether stage 1 unprivileged permission overlay is enabled

boolean AArch64.S1E0POEnabled(Regime regime, bit nv1)
    assert HasUnprivileged(regime);

    if !HaveS1POExt() then
        return FALSE;

    case regime of
        when Regime_EL20 return IsTCR2EL2Enabled() && TCR2_EL2.E0POE == '1';
        when Regime_EL10 return IsTCR2EL1Enabled() && nv1 == '0' && TCR2_EL1.E0POE == '1';
```

Library pseudocode for aarch64/translation/vmsa_walkparams/AArch64.S1EPD

```
// AArch64.S1EPD()
// ===============
// Determine whether stage 1 translation table walk is allowed for the VA range

bit AArch64.S1EPD(Regime regime, bits(64) va)
    assert HasUnprivileged(regime);
    varange = AArch64.GetVARange(va);

    case regime of
        when Regime_EL20 return if varange == VARange_LOWER then TCR_EL2.EPD0 else TCR_EL2.EPD1;
        when Regime_EL10 return if varange == VARange_LOWER then TCR_EL1.EPD0 else TCR_EL1.EPD1;
```

Library pseudocode for aarch64/translation/vmsa_walkparams/AArch64.S1Enabled

```
// AArch64.S1Enabled()
// ===================
// Determine if stage 1 is enabled for the access type for this translation regime

boolean AArch64.S1Enabled(Regime regime, AccessType acctype)
    case regime of
        when Regime_EL3  return SCTLR_EL3.M == '1';
        when Regime_EL2  return SCTLR_EL2.M == '1';
        when Regime_EL20 return SCTLR_EL2.M == '1';
        when Regime_EL10 return (!EL2Enabled() || HCR_EL2.<DC,TGE> == '00') && SCTLR_EL1.M == '1';
```

Library pseudocode for aarch64/translation/vmsa_walkparams/AArch64.S1ICacheEnabled

```
// AArch64.S1ICacheEnabled()
// =========================
// Determine cacheability of stage 1 instruction fetches

boolean AArch64.S1ICacheEnabled(Regime regime)
    case regime of
        when Regime_EL3  return SCTLR_EL3.I == '1';
        when Regime_EL2  return SCTLR_EL2.I == '1';
        when Regime_EL20 return SCTLR_EL2.I == '1';
        when Regime_EL10 return SCTLR_EL1.I == '1';
```

Library pseudocode for aarch64/translation/vmsa_walkparams/AArch64.S1MinTxSZ

```
// AArch64.S1MinTxSZ()
// ===================
// Retrieve the minimum value of TxSZ indicating maximum input address size for stage 1

integer AArch64.S1MinTxSZ(Regime regime, bit d128, bit ds, TGx tgx)
    if Have56BitVAExt() && d128 == '1' then
        if HasUnprivileged(regime) then
            return 9;
        else
            return 8;
    if (Have52BitVAExt() && tgx == TGx_64KB) || ds == '1' then
        return 12;

    return 16;
```

Library pseudocode for aarch64/translation/vmsa_walkparams/AArch64.S1POEnabled

```
// AArch64.S1POEnabled()
// =====================
// Determine whether stage 1 privileged permission overlay is enabled

boolean AArch64.S1POEnabled(Regime regime)
    if !HaveS1POExt() then
        return FALSE;

    case regime of
        when Regime_EL3  return TCR_EL3.POE == '1';
        when Regime_EL2  return IsTCR2EL2Enabled() && TCR2_EL2.POE == '1';
        when Regime_EL20 return IsTCR2EL2Enabled() && TCR2_EL2.POE == '1';
        when Regime_EL10 return IsTCR2EL1Enabled() && TCR2_EL1.POE == '1';
```

Library pseudocode for aarch64/translation/vmsa_walkparams/AArch64.S1POR

```
// AArch64.S1POR()
// ===============
// Identify stage 1 permissions overlay register for the acting translation regime

S1PORType AArch64.S1POR(Regime regime)
    case regime of
        when Regime_EL3  return POR_EL3;
        when Regime_EL2  return POR_EL2;
        when Regime_EL20 return POR_EL2;
        when Regime_EL10 return POR_EL1;
```

Library pseudocode for aarch64/translation/vmsa_walkparams/AArch64.S1TTBR

```
// AArch64.S1TTBR()
// ================
// Identify stage 1 table base register for the acting translation regime

bits(128) AArch64.S1TTBR(Regime regime, bits(64) va)
    varange = AArch64.GetVARange(va);

    case regime of
        when Regime_EL3  return ZeroExtend(TTBR0_EL3, 128);
        when Regime_EL2  return ZeroExtend(TTBR0_EL2, 128);
        when Regime_EL20
            if varange == VARange_LOWER then
                return ZeroExtend(TTBR0_EL2, 128);
            else
                return ZeroExtend(TTBR1_EL2, 128);
        when Regime_EL10
            if varange == VARange_LOWER then
                return ZeroExtend(TTBR0_EL1, 128);
            else
                return ZeroExtend(TTBR1_EL1, 128);
```

Library pseudocode for aarch64/translation/vmsa_walkparams/AArch64.S1TTWParamsEL10

```
// AArch64.S1TTWParamsEL10()
// =========================
// Gather stage 1 translation table walk parameters for EL1&0 regime
// (with EL2 enabled or disabled)

S1TTWParams AArch64.S1TTWParamsEL10(VARange varange)
    S1TTWParams walkparams;

    if Have128BitDescriptorExt() && IsTCR2EL1Enabled() then
        walkparams.d128 = TCR2_EL1.D128;
    else
        walkparams.d128 = '0';
    if varange == VARange_LOWER then
        walkparams.tgx   = AArch64.S1DecodeTG0(TCR_EL1.TG0);
        walkparams.txsz  = TCR_EL1.T0SZ;
        walkparams.irgn  = TCR_EL1.IRGN0;
        walkparams.orgn  = TCR_EL1.ORGN0;
        walkparams.sh    = TCR_EL1.SH0;
        walkparams.tbi   = TCR_EL1.TBI0;

        walkparams.nfd   = if HaveSVE() || HaveTME() then TCR_EL1.NFD0  else '0';
        walkparams.tbid  = if HavePACExt()           then TCR_EL1.TBID0 else '0';
        walkparams.e0pd  = if HaveE0PDExt()          then TCR_EL1.E0PD0 else '0';
        walkparams.hpd   = if AArch64.HaveHPDExt()   then TCR_EL1.HPD0  else '0';
        walkparams.mtx   = if HaveMTE4Ext() then TCR_EL1.MTX0 else '0';
        walkparams.skl   = if walkparams.d128 == '1' then TTBR0_EL1.SKL else '00';
        walkparams.disch = if walkparams.d128 == '1' then TCR2_EL1.DisCH0 else '0';
    else
        walkparams.tgx   = AArch64.S1DecodeTG1(TCR_EL1.TG1);
        walkparams.txsz  = TCR_EL1.T1SZ;
        walkparams.irgn  = TCR_EL1.IRGN1;
        walkparams.orgn  = TCR_EL1.ORGN1;
        walkparams.sh    = TCR_EL1.SH1;
        walkparams.tbi   = TCR_EL1.TBI1;

        walkparams.nfd   = if HaveSVE() || HaveTME() then TCR_EL1.NFD1  else '0';
        walkparams.tbid  = if HavePACExt()           then TCR_EL1.TBID1 else '0';
        walkparams.e0pd  = if HaveE0PDExt()          then TCR_EL1.E0PD1 else '0';
        walkparams.hpd   = if AArch64.HaveHPDExt()   then TCR_EL1.HPD1  else '0';
        walkparams.mtx   = if HaveMTE4Ext()          then TCR_EL1.MTX1 else '0';
        walkparams.skl   = if walkparams.d128 == '1' then TTBR1_EL1.SKL else '00';
        walkparams.disch = if walkparams.d128 == '1' then TCR2_EL1.DisCH1 else '0';

    walkparams.mair = MAIR_EL1;
    if HaveAIEExt() then
        walkparams.mair2 = MAIR2_EL1;
    walkparams.aie  = if HaveAIEExt() && IsTCR2EL1Enabled() then TCR2_EL1.AIE else '0';
    walkparams.wxn  = SCTLR_EL1.WXN;
    walkparams.ps   = TCR_EL1.IPS;
    walkparams.ee   = SCTLR_EL1.EE;
    if (HaveEL(EL3) && (!HaveRME() || HaveSecureEL2Ext())) then
        walkparams.sif = SCR_EL3.SIF;
    else
        walkparams.sif = '0';

    if EL2Enabled() then
        walkparams.dc  = HCR_EL2.DC;
        walkparams.dct = if HaveMTE2Ext() then HCR_EL2.DCT else '0';

    if HaveTrapLoadStoreMultipleDeviceExt() then
        walkparams.ntlsmd = SCTLR_EL1.nTLSMD;
    else
        walkparams.ntlsmd = '1';

    if EL2Enabled() then
        if HCR_EL2.<NV,NV1> == '01' then
            case ConstrainUnpredictable(Unpredictable_NVNV1) of
                when Constraint_NVNV1_00 walkparams.nv1 = '0';
                when Constraint_NVNV1_01 walkparams.nv1 = '1';
                when Constraint_NVNV1_11 walkparams.nv1 = '1';
```

```
        else
            walkparams.nv1 = HCR_EL2.NV1;
    else
        walkparams.nv1 = '0';

    walkparams.cmow = if HaveFeatCMOW()             then SCTLR_EL1.CMOW else '0';
    walkparams.ha   = if HaveAccessFlagUpdateExt() then TCR_EL1.HA else '0';
    walkparams.hd   = if HaveDirtyBitModifierExt() then TCR_EL1.HD else '0';
    if walkparams.tgx IN {TGx_4KB, TGx_16KB} && Have52BitIPAAndPASpaceExt() then
        walkparams.ds = TCR_EL1.DS;
    else
        walkparams.ds = '0';
    if walkparams.d128 == '1' then
        walkparams.pie = '1';
    else
        walkparams.pie = if HaveS1PIExt() && IsTCR2EL1Enabled() then TCR2_EL1.PIE else '0';
    if HaveS1PIExt() then
        walkparams.pir = PIR_EL1;
        if walkparams.nv1 != '1'then
            walkparams.pire0 = PIRE0_EL1;
    if HavePAN3Ext() then
        walkparams.epan = if walkparams.pie == '0' then SCTLR_EL1.EPAN else '1';
    else
        walkparams.epan = '0';
    if HaveTHExt() && walkparams.d128 == '0' && IsTCR2EL1Enabled() then
        walkparams.pnch = TCR2_EL1.PnCH;
    else
        walkparams.pnch = '0';
    if HaveAccessFlagUpdateForTableExt() && walkparams.ha == '1' && IsTCR2EL1Enabled() then
        walkparams.haft = TCR2_EL1.HAFT;
    else
        walkparams.haft = '0';
    walkparams.emec = if HaveFeatMEC() && IsSCTLR2EL2Enabled() then SCTLR2_EL2.EMEC else '0';

    return walkparams;
```

Library pseudocode for aarch64/translation/vmsa_walkparams/AArch64.S1TTWParamsEL2

```
// AArch64.S1TTWParamsEL2()
// ========================
// Gather stage 1 translation table walk parameters for EL2 regime

S1TTWParams AArch64.S1TTWParamsEL2(SecurityState ss)
    S1TTWParams walkparams;

    walkparams.tgx  = AArch64.S1DecodeTG0(TCR_EL2.TG0);
    walkparams.txsz = TCR_EL2.T0SZ;
    walkparams.ps   = TCR_EL2.PS;
    walkparams.irgn = TCR_EL2.IRGN0;
    walkparams.orgn = TCR_EL2.ORGN0;
    walkparams.sh   = TCR_EL2.SH0;
    walkparams.tbi  = TCR_EL2.TBI;
    walkparams.mair = MAIR_EL2;
    if HaveAIEExt() then
        walkparams.mair2 = MAIR2_EL2;
    walkparams.aie  = if HaveAIEExt() && IsTCR2EL2Enabled() then TCR2_EL2.AIE else '0';
    walkparams.wxn  = SCTLR_EL2.WXN;
    walkparams.ee   = SCTLR_EL2.EE;
    if (HaveEL(EL3) && (!HaveRME() || HaveSecureEL2Ext())) then
        walkparams.sif = SCR_EL3.SIF;
    else
        walkparams.sif = '0';

    walkparams.tbid = if HavePACExt()              then TCR_EL2.TBID else '0';
    walkparams.hpd  = if AArch64.HaveHPDExt()      then TCR_EL2.HPD  else '0';
    walkparams.ha   = if HaveAccessFlagUpdateExt() then TCR_EL2.HA   else '0';
    walkparams.hd   = if HaveDirtyBitModifierExt() then TCR_EL2.HD   else '0';
    if walkparams.tgx IN {TGx_4KB, TGx_16KB} && Have52BitIPAAndPASpaceExt() then
        walkparams.ds = TCR_EL2.DS;
    else
        walkparams.ds = '0';
    walkparams.pie = if HaveS1PIExt() && IsTCR2EL2Enabled() then TCR2_EL2.PIE else '0';
    if HaveS1PIExt() then
        walkparams.pir = PIR_EL2;
    walkparams.mtx  = if HaveMTE4Ext()             then TCR_EL2.MTX else '0';
    walkparams.pnch = if HaveTHExt() && IsTCR2EL2Enabled() then TCR2_EL2.PnCH else '0';
    if HaveAccessFlagUpdateForTableExt() && walkparams.ha == '1' && IsTCR2EL2Enabled() then
        walkparams.haft = TCR2_EL2.HAFT;
    else
        walkparams.haft = '0';
    walkparams.emec = if HaveFeatMEC() && IsSCTLR2EL2Enabled() then SCTLR2_EL2.EMEC else '0';
    if HaveFeatMEC() && ss == SS_Realm && IsTCR2EL2Enabled() then
        walkparams.amec = TCR2_EL2.AMEC0;
    else
        walkparams.amec = '0';

    return walkparams;
```

Library pseudocode for aarch64/translation/vmsa_walkparams/AArch64.S1TTWParamsEL20

```
// AArch64.S1TTWParamsEL20()
// =========================
// Gather stage 1 translation table walk parameters for EL2&0 regime

S1TTWParams AArch64.S1TTWParamsEL20(SecurityState ss, VARange varange)
    S1TTWParams walkparams;

    if Have128BitDescriptorExt() && IsTCR2EL2Enabled() then
        walkparams.d128 = TCR2_EL2.D128;
    else
        walkparams.d128 = '0';
    if varange == VARange_LOWER then
        walkparams.tgx   = AArch64.S1DecodeTG0(TCR_EL2.TG0);
        walkparams.txsz  = TCR_EL2.T0SZ;
        walkparams.irgn  = TCR_EL2.IRGN0;
        walkparams.orgn  = TCR_EL2.ORGN0;
        walkparams.sh    = TCR_EL2.SH0;
        walkparams.tbi   = TCR_EL2.TBI0;

        walkparams.nfd   = if HaveSVE() || HaveTME() then TCR_EL2.NFD0  else '0';
        walkparams.tbid  = if HavePACExt()           then TCR_EL2.TBID0 else '0';
        walkparams.e0pd  = if HaveE0PDExt()          then TCR_EL2.E0PD0 else '0';
        walkparams.hpd   = if AArch64.HaveHPDExt()   then TCR_EL2.HPD0  else '0';
        walkparams.mtx   = if HaveMTE4Ext()          then TCR_EL2.MTX0 else '0';
        walkparams.skl   = if walkparams.d128 == '1' then TTBR0_EL2.SKL else '00';
        walkparams.disch = if walkparams.d128 == '1' then TCR2_EL2.DisCH0 else '0';
    else
        walkparams.tgx   = AArch64.S1DecodeTG1(TCR_EL2.TG1);
        walkparams.txsz  = TCR_EL2.T1SZ;
        walkparams.irgn  = TCR_EL2.IRGN1;
        walkparams.orgn  = TCR_EL2.ORGN1;
        walkparams.sh    = TCR_EL2.SH1;
        walkparams.tbi   = TCR_EL2.TBI1;

        walkparams.nfd   = if HaveSVE() || HaveTME() then TCR_EL2.NFD1  else '0';
        walkparams.tbid  = if HavePACExt()           then TCR_EL2.TBID1 else '0';
        walkparams.e0pd  = if HaveE0PDExt()          then TCR_EL2.E0PD1 else '0';
        walkparams.hpd   = if AArch64.HaveHPDExt()   then TCR_EL2.HPD1  else '0';
        walkparams.mtx   = if HaveMTE4Ext()          then TCR_EL2.MTX1 else '0';
        walkparams.skl   = if walkparams.d128 == '1' then TTBR1_EL2.SKL else '00';
        walkparams.disch = if walkparams.d128 == '1' then TCR2_EL2.DisCH1 else '0';
    walkparams.mair = MAIR_EL2;
    if HaveAIEExt() then
        walkparams.mair2 = MAIR2_EL2;
    walkparams.aie  = if HaveAIEExt() && IsTCR2EL2Enabled() then TCR2_EL2.AIE else '0';
    walkparams.wxn  = SCTLR_EL2.WXN;
    walkparams.ps   = TCR_EL2.IPS;
    walkparams.ee   = SCTLR_EL2.EE;
    if (HaveEL(EL3) && (!HaveRME() || HaveSecureEL2Ext())) then
        walkparams.sif = SCR_EL3.SIF;
    else
        walkparams.sif = '0';

    if HaveTrapLoadStoreMultipleDeviceExt() then
        walkparams.ntlsmd = SCTLR_EL2.nTLSMD;
    else
        walkparams.ntlsmd = '1';

    walkparams.cmow = if HaveFeatCMOW()            then SCTLR_EL2.CMOW else '0';
    walkparams.ha   = if HaveAccessFlagUpdateExt() then TCR_EL2.HA else '0';
    walkparams.hd   = if HaveDirtyBitModifierExt() then TCR_EL2.HD else '0';
    if walkparams.tgx IN {TGx_4KB, TGx_16KB} && Have52BitIPAAndPASpaceExt() then
        walkparams.ds = TCR_EL2.DS;
    else
        walkparams.ds = '0';
    if walkparams.d128 == '1' then
        walkparams.pie = '1';
    else
        walkparams.pie = if HaveS1PIExt() && IsTCR2EL2Enabled() then TCR2_EL2.PIE else '0';
    if HaveS1PIExt() then
```

```
        walkparams.pir   = PIR_EL2;
        walkparams.pire0 = PIRE0_EL2;
    if HavePAN3Ext() then
        walkparams.epan = if walkparams.pie == '0' then SCTLR_EL2.EPAN else '1';
    else
        walkparams.epan = '0';
    if HaveTHExt() && walkparams.d128 == '0' && IsTCR2EL2Enabled() then
        walkparams.pnch = TCR2_EL2.PnCH;
    else
        walkparams.pnch = '0';
    if HaveAccessFlagUpdateForTableExt() && walkparams.ha == '1' && IsTCR2EL2Enabled() then
        walkparams.haft = TCR2_EL2.HAFT;
    else
        walkparams.haft = '0';
    walkparams.emec = if HaveFeatMEC() && IsSCTLR2EL2Enabled() then SCTLR2_EL2.EMEC else '0';
    if HaveFeatMEC() && ss == SS_Realm && IsTCR2EL2Enabled() then
        walkparams.amec = if varange == VARange_LOWER then TCR2_EL2.AMEC0 else TCR2_EL2.AMEC1;
    else
        walkparams.amec = '0';

    return walkparams;
```

Library pseudocode for aarch64/translation/vmsa_walkparams/AArch64.S1TTWParamsEL3

```
// AArch64.S1TTWParamsEL3()
// ========================
// Gather stage 1 translation table walk parameters for EL3 regime

S1TTWParams AArch64.S1TTWParamsEL3()
    S1TTWParams walkparams;

    walkparams.tgx  = AArch64.S1DecodeTG0(TCR_EL3.TG0);
    walkparams.txsz = TCR_EL3.T0SZ;
    walkparams.ps   = TCR_EL3.PS;
    walkparams.irgn = TCR_EL3.IRGN0;
    walkparams.orgn = TCR_EL3.ORGN0;
    walkparams.sh   = TCR_EL3.SH0;
    walkparams.tbi  = TCR_EL3.TBI;
    walkparams.mair = MAIR_EL3;
    if HaveAIEExt() then
        walkparams.mair2 = MAIR2_EL3;
    walkparams.aie  = if HaveAIEExt() then TCR_EL3.AIE else '0';
    walkparams.wxn  = SCTLR_EL3.WXN;
    walkparams.ee   = SCTLR_EL3.EE;
    walkparams.sif = if !HaveRME() || HaveSecureEL2Ext() then SCR_EL3.SIF else '0';

    walkparams.tbid = if HavePACExt()              then TCR_EL3.TBID else '0';
    walkparams.hpd  = if AArch64.HaveHPDExt()      then TCR_EL3.HPD  else '0';
    walkparams.ha   = if HaveAccessFlagUpdateExt() then TCR_EL3.HA   else '0';
    walkparams.hd   = if HaveDirtyBitModifierExt() then TCR_EL3.HD   else '0';
    if walkparams.tgx IN {TGx_4KB, TGx_16KB} && Have52BitIPAAndPASpaceExt() then
        walkparams.ds = TCR_EL3.DS;
    else
        walkparams.ds = '0';
    walkparams.d128  = if Have128BitDescriptorExt() then TCR_EL3.D128 else '0';
    walkparams.skl   = if walkparams.d128 == '1' then TTBR0_EL3.SKL else '00';
    walkparams.disch = if walkparams.d128 == '1' then TCR_EL3.DisCH0 else '0';
    if walkparams.d128 == '1' then
        walkparams.pie = '1';
    else
        walkparams.pie = if HaveS1PIExt() then TCR_EL3.PIE else '0';
    if HaveS1PIExt() then
        walkparams.pir = PIR_EL3;
    walkparams.mtx  = if HaveMTE4Ext()             then TCR_EL3.MTX else '0';
    if HaveTHExt() && walkparams.d128 == '0' then
        walkparams.pnch = TCR_EL3.PnCH;
    else
        walkparams.pnch = '0';
    if HaveAccessFlagUpdateForTableExt() && walkparams.ha == '1' then
        walkparams.haft = TCR_EL3.HAFT;
    else
        walkparams.haft = '0';
    walkparams.emec = if HaveFeatMEC() then SCTLR2_EL3.EMEC else '0';

    return walkparams;
```

Library pseudocode for aarch64/translation/vmsa_walkparams/AArch64.S2DecodeTG0

```
// AArch64.S2DecodeTG0()
// =====================
// Decode stage 2 granule size configuration bits TG0

TGx AArch64.S2DecodeTG0(bits(2) tg0_in)
    bits(2) tg0 = tg0_in;
    TGx tgx;

    if tg0 == '11' then
        tg0 = bits(2) IMPLEMENTATION_DEFINED "TG0 encoded granule size";

    case tg0 of
        when '00'   tgx = TGx_4KB;
        when '01'   tgx = TGx_64KB;
        when '10'   tgx = TGx_16KB;

    if !AArch64.HaveS2TG(tgx) then
        case bits(2) IMPLEMENTATION_DEFINED "TG0 encoded granule size" of
            when '00'   tgx = TGx_4KB;
            when '01'   tgx = TGx_64KB;
            when '10'   tgx = TGx_16KB;

    return tgx;
```

Library pseudocode for aarch64/translation/vmsa_walkparams/AArch64.S2MinTxSZ

```
// AArch64.S2MinTxSZ()
// ===================
// Retrieve the minimum value of TxSZ indicating maximum input address size for stage 2

integer AArch64.S2MinTxSZ(bit d128, bit ds, TGx tgx, boolean s1aarch64)
    ips = AArch64.PAMax();

    if d128 == '0' then
        if Have52BitPAExt() && tgx != TGx_64KB && ds == '0' then
            ips = Min(48, AArch64.PAMax());
        else
            ips = Min(52, AArch64.PAMax());
    min_txsz = 64 - ips;
    if !s1aarch64 then
        // EL1 is AArch32
        min_txsz = Min(min_txsz, 24);

    return min_txsz;
```

Library pseudocode for aarch64/translation/vmsa_walkparams/AArch64.SS2TTWParams

```
// AArch64.SS2TTWParams()
// ======================
// Gather walk parameters specific for secure stage 2 translation

S2TTWParams AArch64.SS2TTWParams(PASpace ipaspace, boolean s1aarch64)
    S2TTWParams walkparams;

    walkparams.d128 = if Have128BitDescriptorExt() then VTCR_EL2.D128 else '0';
    if ipaspace == PAS_Secure then
        walkparams.tgx  = AArch64.S2DecodeTG0(VSTCR_EL2.TG0);
        walkparams.txsz = VSTCR_EL2.T0SZ;
        if walkparams.d128 == '1' then
            walkparams.skl = VSTTBR_EL2.SKL;
        else
            walkparams.sl0 = VSTCR_EL2.SL0;
        if walkparams.tgx == TGx_4KB && Have52BitIPAAndPASpaceExt() then
            walkparams.sl2 = VSTCR_EL2.SL2 AND VTCR_EL2.DS;
        else
            walkparams.sl2 = '0';
    elsif ipaspace == PAS_NonSecure then
        walkparams.tgx  = AArch64.S2DecodeTG0(VTCR_EL2.TG0);
        walkparams.txsz = VTCR_EL2.T0SZ;
        if walkparams.d128 == '1' then
            walkparams.skl = VTTBR_EL2.SKL;
        else
            walkparams.sl0 = VTCR_EL2.SL0;
        if walkparams.tgx == TGx_4KB && Have52BitIPAAndPASpaceExt() then
            walkparams.sl2 = VTCR_EL2.SL2 AND VTCR_EL2.DS;
        else
            walkparams.sl2 = '0';
    else
        Unreachable();

    walkparams.sw   = VSTCR_EL2.SW;
    walkparams.nsw  = VTCR_EL2.NSW;
    walkparams.sa   = VSTCR_EL2.SA;
    walkparams.nsa  = VTCR_EL2.NSA;
    walkparams.vm   = HCR_EL2.VM OR HCR_EL2.DC;
    walkparams.ps   = VTCR_EL2.PS;
    walkparams.irgn = VTCR_EL2.IRGN0;
    walkparams.orgn = VTCR_EL2.ORGN0;
    walkparams.sh   = VTCR_EL2.SH0;
    walkparams.ee   = SCTLR_EL2.EE;

    walkparams.ptw = if HCR_EL2.TGE == '0'          then HCR_EL2.PTW else '0';
    walkparams.fwb = if HaveStage2MemAttrControl() then HCR_EL2.FWB else '0';
    walkparams.ha  = if HaveAccessFlagUpdateExt()  then VTCR_EL2.HA else '0';
    walkparams.hd  = if HaveDirtyBitModifierExt()  then VTCR_EL2.HD else '0';
    if walkparams.tgx IN {TGx_4KB, TGx_16KB} && Have52BitIPAAndPASpaceExt() then
        walkparams.ds = VTCR_EL2.DS;
    else
        walkparams.ds = '0';
    walkparams.cmow = if HaveFeatCMOW() && IsHCRXEL2Enabled() then HCRX_EL2.CMOW else '0';
    if walkparams.d128 == '1' then
        walkparams.s2pie = '1';
    else
        walkparams.s2pie = if HaveS2PIExt() then VTCR_EL2.S2PIE else '0';
    walkparams.s2pir = if HaveS2PIExt() then S2PIR_EL2 else Zeros(64);
    if HaveTHExt() && walkparams.d128 != '1' then
        walkparams.assuredonly = VTCR_EL2.AssuredOnly;
    else
        walkparams.assuredonly = '0';
    walkparams.tl0   = if HaveTHExt() then VTCR_EL2.TL0 else '0';
    walkparams.tl1   = if HaveTHExt() then VTCR_EL2.TL1 else '0';
    if HaveAccessFlagUpdateForTableExt() && walkparams.ha == '1' then
        walkparams.haft = VTCR_EL2.HAFT;
    else
        walkparams.haft = '0';
    walkparams.emec = '0';
```

```
    return walkparams;
```

Library pseudocode for shared/debug/ClearStickyErrors/ClearStickyErrors

```
// ClearStickyErrors()
// ===================

ClearStickyErrors()
    EDSCR.TXU = '0';            // Clear TX underrun flag
    EDSCR.RXO = '0';            // Clear RX overrun flag

    if Halted() then            // in Debug state
        EDSCR.ITO = '0';        // Clear ITR overrun flag

    // If halted and the ITR is not empty then it is UNPREDICTABLE whether the EDSCR.ERR is cleared.
    // The UNPREDICTABLE behavior also affects the instructions in flight, but this is not described
    // in the pseudocode.
    if (Halted() && EDSCR.ITE == '0' &&
          ConstrainUnpredictableBool(Unpredictable_CLEARERRITEZERO)) then
        return;
    EDSCR.ERR = '0';            // Clear cumulative error flag

    return;
```

Library pseudocode for shared/debug/DebugTarget/DebugTarget

```
// DebugTarget()
// =============
// Returns the debug exception target Exception level

bits(2) DebugTarget()
    ss = CurrentSecurityState();
    return DebugTargetFrom(ss);
```

Library pseudocode for shared/debug/DebugTarget/DebugTargetFrom

```
// DebugTargetFrom()
// =================

bits(2) DebugTargetFrom(SecurityState from_state)
    boolean route_to_el2;
    if HaveEL(EL2) && (from_state != SS_Secure ||
        (HaveSecureEL2Ext() && (!HaveEL(EL3) || SCR_EL3.EEL2 == '1'))) then
        if ELUsingAArch32(EL2) then
            route_to_el2 = (HDCR.TDE == '1' || HCR.TGE == '1');
        else
            route_to_el2 = (MDCR_EL2.TDE == '1' || HCR_EL2.TGE == '1');
    else
        route_to_el2 = FALSE;

    bits(2) target;
    if route_to_el2 then
        target = EL2;
    elsif HaveEL(EL3) && !HaveAArch64() && from_state == SS_Secure then
        target = EL3;
    else
        target = EL1;

    return target;
```

Library pseudocode for shared/debug/DoubleLockStatus/DoubleLockStatus

```
// DoubleLockStatus()
// ==================
// Returns the state of the OS Double Lock.
//    FALSE if OSDLR_EL1.DLK == 0 or DBGPRCR_EL1.CORENPDRQ == 1 or the PE is in Debug state.
//    TRUE if OSDLR_EL1.DLK == 1 and DBGPRCR_EL1.CORENPDRQ == 0 and the PE is in Non-debug state.

boolean DoubleLockStatus()
    if !HaveDoubleLock() then
        return FALSE;
    elsif ELUsingAArch32(EL1) then
        return DBGOSDLR.DLK == '1' && DBGPRCR.CORENPDRQ == '0' && !Halted();
    else
        return OSDLR_EL1.DLK == '1' && DBGPRCR_EL1.CORENPDRQ == '0' && !Halted();
```

Library pseudocode for shared/debug/OSLockStatus/OSLockStatus

```
// OSLockStatus()
// ==============
// Returns the state of the OS Lock.

boolean OSLockStatus()
    return (if ELUsingAArch32(EL1) then DBGOSLSR.OSLK else OSLSR_EL1.OSLK) == '1';
```

Library pseudocode for shared/debug/SoftwareLockStatus/Component

```
// Component
// =========
// Component Types.

enumeration Component {
        Component_PMU,
        Component_Debug,
        Component_CTI
};
```

Library pseudocode for shared/debug/SoftwareLockStatus/GetAccessComponent

```
// GetAccessComponent()
// ====================
// Returns the accessed component.

Component GetAccessComponent();
```

Library pseudocode for shared/debug/SoftwareLockStatus/SoftwareLockStatus

```
// SoftwareLockStatus()
// ====================
// Returns the state of the Software Lock.

boolean SoftwareLockStatus()
    Component component = GetAccessComponent();
    if !HaveSoftwareLock(component) then
        return FALSE;
    case component of
        when Component_Debug
            return EDLSR.SLK == '1';
        when Component_PMU
            return PMLSR.SLK == '1';
        when Component_CTI
            return CTILSR.SLK == '1';
        otherwise
            Unreachable();
```

Library pseudocode for shared/debug/authentication/AccessState

```
// AccessState()
// =============
// Returns the Security state of the access.

SecurityState AccessState();
```

Library pseudocode for shared/debug/authentication/AllowExternalDebugAccess

```
// AllowExternalDebugAccess()
// ==========================
// Returns TRUE if an external debug interface access to the External debug registers
// is allowed, FALSE otherwise.

boolean AllowExternalDebugAccess()
    // The access may also be subject to OS Lock, power-down, etc.
    return AllowExternalDebugAccess(AccessState());

// AllowExternalDebugAccess()
// ==========================
// Returns TRUE if an external debug interface access to the External debug registers
// is allowed for the given Security state, FALSE otherwise.

boolean AllowExternalDebugAccess(SecurityState access_state)
    // The access may also be subject to OS Lock, power-down, etc.
    if HaveRME() then
        case MDCR_EL3.<EDADE,EDAD> of
            when '00' return TRUE;
            when '01' return access_state IN {SS_Root, SS_Secure};
            when '10' return access_state IN {SS_Root, SS_Realm};
            when '11' return access_state == SS_Root;

    if HaveSecureExtDebugView() then
        if access_state == SS_Secure then return TRUE;
    else
        if !ExternalInvasiveDebugEnabled() then return FALSE;
        if ExternalSecureInvasiveDebugEnabled() then return TRUE;

    if HaveEL(EL3) then
        EDAD_bit = if ELUsingAArch32(EL3) then SDCR.EDAD else MDCR_EL3.EDAD;
        return EDAD_bit == '0';
    else
        return NonSecureOnlyImplementation();
```

Library pseudocode for shared/debug/authentication/AllowExternalPMUAccess

```
// AllowExternalPMUAccess()
// ========================
// Returns TRUE if an external debug interface access to the PMU registers is
// allowed, FALSE otherwise.

boolean AllowExternalPMUAccess()
    // The access may also be subject to OS Lock, power-down, etc.
    return AllowExternalPMUAccess(AccessState());

// AllowExternalPMUAccess()
// ========================
// Returns TRUE if an external debug interface access to the PMU registers is
// allowed for the given Security state, FALSE otherwise.

boolean AllowExternalPMUAccess(SecurityState access_state)
    // The access may also be subject to OS Lock, power-down, etc.
    if HaveRME() then
        case MDCR_EL3.<EPMADE,EPMAD> of
            when '00' return TRUE;
            when '01' return access_state IN {SS_Root, SS_Secure};
            when '10' return access_state IN {SS_Root, SS_Realm};
            when '11' return access_state == SS_Root;

    if HaveSecureExtDebugView() then
        if access_state == SS_Secure then return TRUE;
    else
        if !ExternalInvasiveDebugEnabled() then return FALSE;
        if ExternalSecureInvasiveDebugEnabled() then return TRUE;

    if HaveEL(EL3) then
        EPMAD_bit = if ELUsingAArch32(EL3) then SDCR.EPMAD else MDCR_EL3.EPMAD;
        return EPMAD_bit == '0';
    else
        return NonSecureOnlyImplementation();
```

Library pseudocode for shared/debug/authentication/AllowExternalTraceAccess

```
// AllowExternalTraceAccess()
// ==========================
// Returns TRUE if an external Trace access to the Trace registers is allowed, FALSE otherwise.

boolean AllowExternalTraceAccess()
    if !HaveTraceBufferExtension() then
        return TRUE;
    else
        return AllowExternalTraceAccess(AccessState());

// AllowExternalTraceAccess()
// ==========================
// Returns TRUE if an external Trace access to the Trace registers is allowed for the
// given Security state, FALSE otherwise.

boolean AllowExternalTraceAccess(SecurityState access_state)
    // The access may also be subject to OS lock, power-down, etc.
    if !HaveTraceBufferExtension() then return TRUE;
    assert HaveSecureExtDebugView();
    if HaveRME() then
        case MDCR_EL3.<ETADE,ETAD> of
            when '00' return TRUE;
            when '01' return access_state IN {SS_Root, SS_Secure};
            when '10' return access_state IN {SS_Root, SS_Realm};
            when '11' return access_state == SS_Root;

    if access_state == SS_Secure then return TRUE;
    if HaveEL(EL3) then
        // External Trace access is not supported for EL3 using AArch32
        assert !ELUsingAArch32(EL3);
        return MDCR_EL3.ETAD == '0';
    else
        return NonSecureOnlyImplementation();
```

Library pseudocode for shared/debug/authentication/Debug_authentication

```
Signal DBGEN;
Signal NIDEN;
Signal SPIDEN;
Signal SPNIDEN;
Signal RLPIDEN;
Signal RTPIDEN;
```

Library pseudocode for shared/debug/authentication/ExternalInvasiveDebugEnabled

```
// ExternalInvasiveDebugEnabled()
// ==============================
// The definition of this function is IMPLEMENTATION DEFINED.
// In the recommended interface, this function returns the state of the DBGEN signal.

boolean ExternalInvasiveDebugEnabled()
    return DBGEN == Signal_High;
```

Library pseudocode for shared/debug/authentication/ExternalNoninvasiveDebugAllowed

```
// ExternalNoninvasiveDebugAllowed()
// =================================
// Returns TRUE if Trace and PC Sample-based Profiling are allowed

boolean ExternalNoninvasiveDebugAllowed()
    return ExternalNoninvasiveDebugAllowed(PSTATE.EL);

// ExternalNoninvasiveDebugAllowed()
// =================================

boolean ExternalNoninvasiveDebugAllowed(bits(2) el)
    if !ExternalNoninvasiveDebugEnabled() then return FALSE;
    ss = SecurityStateAtEL(el);

    if ((ELUsingAArch32(EL3) || ELUsingAArch32(EL1)) && el == EL0 &&
        ss == SS_Secure && SDER.SUNIDEN == '1') then
        return TRUE;

    case ss of
        when SS_NonSecure return TRUE;
        when SS_Secure    return ExternalSecureNoninvasiveDebugEnabled();
        when SS_Realm     return ExternalRealmNoninvasiveDebugEnabled();
        when SS_Root      return ExternalRootNoninvasiveDebugEnabled();
```

Library pseudocode for shared/debug/authentication/ExternalNoninvasiveDebugEnabled

```
// ExternalNoninvasiveDebugEnabled()
// =================================
// This function returns TRUE if the FEAT_Debugv8p4 is implemented.
// Otherwise, this function is IMPLEMENTATION DEFINED, and, in the
// recommended interface, ExternalNoninvasiveDebugEnabled returns
// the state of the (DBGEN OR NIDEN) signal.

boolean ExternalNoninvasiveDebugEnabled()
    return !HaveNoninvasiveDebugAuth() || ExternalInvasiveDebugEnabled() || NIDEN == Signal_High;
```

Library pseudocode for shared/debug/authentication/ExternalRealmInvasiveDebugEnabled

```
// ExternalRealmInvasiveDebugEnabled()
// ===================================
// The definition of this function is IMPLEMENTATION DEFINED.
// In the recommended interface, this function returns the state of the
// (DBGEN AND RLPIDEN) signal.

boolean ExternalRealmInvasiveDebugEnabled()
    if !HaveRME() then return FALSE;
    return ExternalInvasiveDebugEnabled() && RLPIDEN == Signal_High;
```

Library pseudocode for shared/debug/authentication/ExternalRealmNoninvasiveDebugEnabled

```
// ExternalRealmNoninvasiveDebugEnabled()
// ======================================
// The definition of this function is IMPLEMENTATION DEFINED.
// In the recommended interface, this function returns the state of the
// (DBGEN AND RLPIDEN) signal.

boolean ExternalRealmNoninvasiveDebugEnabled()
    if !HaveRME() then return FALSE;
    return ExternalRealmInvasiveDebugEnabled();
```

Library pseudocode for shared/debug/authentication/ExternalRootInvasiveDebugEnabled

```
// ExternalRootInvasiveDebugEnabled()
// ==================================
// The definition of this function is IMPLEMENTATION DEFINED.
// In the recommended interface, this function returns the state of the
// (DBGEN AND RLPIDEN AND RTPIDEN AND SPIDEN) signal when FEAT_SEL2 is implemented
// and the (DBGEN AND RLPIDEN AND RTPIDEN) signal when FEAT_SEL2 is not implemented.

boolean ExternalRootInvasiveDebugEnabled()
    if !HaveRME() then return FALSE;
    return (ExternalInvasiveDebugEnabled() &&
            (!HaveSecureEL2Ext() || ExternalSecureInvasiveDebugEnabled()) &&
            ExternalRealmInvasiveDebugEnabled() &&
            RTPIDEN == Signal_High);
```

Library pseudocode for shared/debug/authentication/ExternalRootNoninvasiveDebugEnabled

```
// ExternalRootNoninvasiveDebugEnabled()
// =====================================
// The definition of this function is IMPLEMENTATION DEFINED.
// In the recommended interface, this function returns the state of the
// (DBGEN AND RLPIDEN AND SPIDEN AND RTPIDEN) signal.

boolean ExternalRootNoninvasiveDebugEnabled()
    if !HaveRME() then return FALSE;
    return ExternalRootInvasiveDebugEnabled();
```

Library pseudocode for shared/debug/authentication/ExternalSecureInvasiveDebugEnabled

```
// ExternalSecureInvasiveDebugEnabled()
// ====================================
// The definition of this function is IMPLEMENTATION DEFINED.
// In the recommended interface, this function returns the state of the (DBGEN AND SPIDEN) signal.
// CoreSight allows asserting SPIDEN without also asserting DBGEN, but this is not recommended.

boolean ExternalSecureInvasiveDebugEnabled()
    if !HaveEL(EL3) && !SecureOnlyImplementation() then return FALSE;
    return ExternalInvasiveDebugEnabled() && SPIDEN == Signal_High;
```

Library pseudocode for shared/debug/authentication/ExternalSecureNoninvasiveDebugEnabled

```
// ExternalSecureNoninvasiveDebugEnabled()
// =======================================
// This function returns the value of ExternalSecureInvasiveDebugEnabled() when FEAT_Debugv8p4
// is implemented. Otherwise, the definition of this function is IMPLEMENTATION DEFINED.
// In the recommended interface, this function returns the state of the (DBGEN OR NIDEN) AND
// (SPIDEN OR SPNIDEN) signal.

boolean ExternalSecureNoninvasiveDebugEnabled()
    if !HaveEL(EL3) && !SecureOnlyImplementation() then return FALSE;
    if HaveNoninvasiveDebugAuth() then
        return (ExternalNoninvasiveDebugEnabled() &&
                (SPIDEN == Signal_High || SPNIDEN == Signal_High));
    else
        return ExternalSecureInvasiveDebugEnabled();
```

Library pseudocode for shared/debug/authentication/IsAccessSecure

```
// IsAccessSecure()
// ================
// Returns TRUE when an access is Secure

boolean IsAccessSecure();
```

Library pseudocode for shared/debug/authentication/IsCorePowered

```
// IsCorePowered()
// ===============
// Returns TRUE if the Core power domain is powered on, FALSE otherwise.

boolean IsCorePowered();
```

Library pseudocode for shared/debug/breakpoint/CheckValidStateMatch

```
// CheckValidStateMatch()
// ======================
// Checks for an invalid state match that will generate Constrained
// Unpredictable behavior, otherwise returns Constraint_NONE.

(Constraint, bits(2), bit, bit, bits(2)) CheckValidStateMatch(bits(2) ssc_in, bit ssce_in,
                                                              bit hmc_in, bits(2) pxc_in,
                                                              boolean isbreakpnt)
    if !HaveRME() then assert ssce_in == '0';
    boolean reserved = FALSE;
    bits(2) ssc = ssc_in;
    bit ssce    = ssce_in;
    bit hmc     = hmc_in;
    bits(2) pxc = pxc_in;

    // Values that are not allocated in any architecture version
    case hmc:ssce:ssc:pxc of
        when '0 0 11 10' reserved = TRUE;
        when '0 0 1x xx' reserved = !HaveSecureState();
        when '1 0 00 x0' reserved = TRUE;
        when '1 0 01 10' reserved = TRUE;
        when '1 0 1x 10' reserved = TRUE;
        when 'x 1 xx xx' reserved = ssc != '01' || (hmc:pxc) IN {'000','110'};
        otherwise        reserved = FALSE;

    // Match 'Usr/Sys/Svc' valid only for AArch32 breakpoints
    if (!isbreakpnt || !HaveAArch32EL(EL1)) && hmc:pxc == '000' && ssc != '11' then
        reserved = TRUE;

    // Both EL3 and EL2 are not implemented
    if !HaveEL(EL3) && !HaveEL(EL2) && (hmc != '0' || ssc != '00') then
        reserved = TRUE;

    // EL3 is not implemented
    if !HaveEL(EL3) && ssc IN {'01','10'} && hmc:ssc:pxc != '10100' then
        reserved = TRUE;

    // EL3 using AArch64 only
    if (!HaveEL(EL3) || !HaveAArch64()) && hmc:ssc:pxc == '11000' then
        reserved = TRUE;

    // EL2 is not implemented
    if !HaveEL(EL2) && hmc:ssc:pxc == '11100' then
        reserved = TRUE;

    // Secure EL2 is not implemented
    if !HaveSecureEL2Ext() && (hmc:ssc:pxc)  IN {'01100','10100','x11x1'} then
        reserved = TRUE;

    if reserved then
        // If parameters are set to a reserved type, behaves as either disabled or a defined type
        Constraint c;
        (c, <hmc,ssc,ssce,pxc>) = ConstrainUnpredictableBits(Unpredictable_RESBPWPCTRL, 6);
        assert c IN {Constraint_DISABLED, Constraint_UNKNOWN};
        if c == Constraint_DISABLED then
            return (c, bits(2) UNKNOWN, bit UNKNOWN, bit UNKNOWN, bits(2) UNKNOWN);
        // Otherwise the value returned by ConstrainUnpredictableBits must be a not-reserved value

    return (Constraint_NONE, ssc, ssce, hmc, pxc);
```

Library pseudocode for shared/debug/breakpoint/ContextMatchingBreakpointRange

```
// ContextMatchingBreakpointRange()
// ================================
// Returns two numbers indicating the index of the first and last context-aware breakpoint.

(integer, integer) ContextMatchingBreakpointRange()
    integer b = NumBreakpointsImplemented();
    integer c = NumContextAwareBreakpointsImplemented();

    if b <= 16 then
        return (b - c, b - 1);
    elsif c <= 16 then
        return (16 - c, 15);
    else
        return (0, c - 1);
```

Library pseudocode for shared/debug/breakpoint/IsContextMatchingBreakpoint

```
// IsContextMatchingBreakpoint()
// =============================
// Returns TRUE if DBGBCR_EL1[n] is a context-aware breakpoint.

boolean IsContextMatchingBreakpoint(integer n)
    (lower, upper) = ContextMatchingBreakpointRange();
    return n >= lower && n <= upper;
```

Library pseudocode for shared/debug/breakpoint/NumBreakpointsImplemented

```
// NumBreakpointsImplemented()
// ===========================
// Returns the number of breakpoints implemented.

integer NumBreakpointsImplemented()
    return integer IMPLEMENTATION_DEFINED "Number of breakpoints";
```

Library pseudocode for shared/debug/breakpoint/NumContextAwareBreakpointsImplemented

```
// NumContextAwareBreakpointsImplemented()
// =======================================
// Returns the number of context-aware breakpoints implemented.

integer NumContextAwareBreakpointsImplemented()
    return integer IMPLEMENTATION_DEFINED "Number of context-aware breakpoints";
```

Library pseudocode for shared/debug/breakpoint/NumWatchpointsImplemented

```
// NumWatchpointsImplemented()
// ===========================
// Returns the number of watchpoints implemented.

integer NumWatchpointsImplemented()
    return integer IMPLEMENTATION_DEFINED "Number of watchpoints";
```

Library pseudocode for shared/debug/cti/CTI_ProcessEvent

```
// CTI_ProcessEvent()
// ==================
// Process a discrete event on a Cross Trigger output event trigger.

CTI_ProcessEvent(CrossTriggerOut id);
```

Library pseudocode for shared/debug/cti/CTI_SetEventLevel

```
// CTI_SetEventLevel()
// ===================
// Set a Cross Trigger multi-cycle input event trigger to the specified level.

CTI_SetEventLevel(CrossTriggerIn id, Signal level);
```

Library pseudocode for shared/debug/cti/CTI_SignalEvent

```
// CTI_SignalEvent()
// =================
// Signal a discrete event on a Cross Trigger input event trigger.

CTI_SignalEvent(CrossTriggerIn id);
```

Library pseudocode for shared/debug/cti/CrossTrigger

```
// CrossTrigger
// ============

enumeration CrossTriggerOut {CrossTriggerOut_DebugRequest, CrossTriggerOut_RestartRequest,
                             CrossTriggerOut_IRQ,          CrossTriggerOut_RSVD3,
                             CrossTriggerOut_TraceExtIn0,  CrossTriggerOut_TraceExtIn1,
                             CrossTriggerOut_TraceExtIn2,  CrossTriggerOut_TraceExtIn3};

enumeration CrossTriggerIn  {CrossTriggerIn_CrossHalt,     CrossTriggerIn_PMUOverflow,
                             CrossTriggerIn_RSVD2,         CrossTriggerIn_RSVD3,
                             CrossTriggerIn_TraceExtOut0,  CrossTriggerIn_TraceExtOut1,
                             CrossTriggerIn_TraceExtOut2,  CrossTriggerIn_TraceExtOut3};
```

Library pseudocode for shared/debug/dccanditr/CheckForDCCInterrupts

```
// CheckForDCCInterrupts()
// =======================

CheckForDCCInterrupts()
    commrx = (EDSCR.RXfull == '1');
    commtx = (EDSCR.TXfull == '0');

    // COMMRX and COMMTX support is optional and not recommended for new designs.
    // SetInterruptRequestLevel(InterruptID_COMMRX, if commrx then HIGH else LOW);
    // SetInterruptRequestLevel(InterruptID_COMMTX, if commtx then HIGH else LOW);

    // The value to be driven onto the common COMMIRQ signal.
    boolean commirq;
    if ELUsingAArch32(EL1) then
        commirq = ((commrx && DBGDCCINT.RX == '1') ||
                   (commtx && DBGDCCINT.TX == '1'));
    else
        commirq = ((commrx && MDCCINT_EL1.RX == '1') ||
                   (commtx && MDCCINT_EL1.TX == '1'));
    SetInterruptRequestLevel(InterruptID_COMMIRQ, if commirq then Signal_High else Signal_Low);

    return;
```

Library pseudocode for shared/debug/dccanditr/DBGDTRRX_EL0

```
// DBGDTRRX_EL0[] (external write)
// ===============================
// Called on writes to debug register 0x08C.

DBGDTRRX_EL0[boolean memory_mapped] = bits(32) value

    if EDPRSR<6:5,0> != '001' then                      // Check DLK, OSLK and PU bits
        IMPLEMENTATION_DEFINED "generate error response";
        return;

    if EDSCR.ERR == '1' then return;                    // Error flag set: ignore write

    if EDSCR.RXfull == '1' || (Halted() && EDSCR.MA == '1' && EDSCR.ITE == '0') then
        EDSCR.RXO = '1';  EDSCR.ERR = '1';              // Overrun condition: ignore write
        return;

    EDSCR.RXfull = '1';
    DTRRX = value;

    if Halted() && EDSCR.MA == '1' then
        EDSCR.ITE = '0';                            // See comments in EDITR[] (external write)
        if !UsingAArch32() then
            ExecuteA64(0xD5330501<31:0>);               // A64 "MRS X1,DBGDTRRX_EL0"
            ExecuteA64(0xB8004401<31:0>);               // A64 "STR W1,[X0],#4"
            X[1, 64] = bits(64) UNKNOWN;
        else
            ExecuteT32(0xEE10<15:0> /*hw1*/, 0x1E15<15:0> /*hw2*/);  // T32 "MRS R1,DBGDTRRXint"
            ExecuteT32(0xF840<15:0> /*hw1*/, 0x1B04<15:0> /*hw2*/);  // T32 "STR R1,[R0],#4"
            R[1] = bits(32) UNKNOWN;
        // If the store aborts, the Data Abort exception is taken and EDSCR.ERR is set to 1
        if EDSCR.ERR == '1' then
            EDSCR.RXfull = bit UNKNOWN;
            DBGDTRRX_EL0 = bits(64) UNKNOWN;
        else
            // "MRS X1,DBGDTRRX_EL0" calls DBGDTR_EL0[] (read) which clears RXfull.
            assert EDSCR.RXfull == '0';

        EDSCR.ITE = '1';                                // See comments in EDITR[] (external write)
    return;

// DBGDTRRX_EL0[] (external read)
// ==============================

bits(32) DBGDTRRX_EL0[boolean memory_mapped]
    return DTRRX;
```

Library pseudocode for shared/debug/dccanditr/DBGDTRTX_EL0

```
// DBGDTRTX_EL0[] (external read)
// ==============================
// Called on reads of debug register 0x080.

bits(32) DBGDTRTX_EL0[boolean memory_mapped]

    if EDPRSR<6:5,0> != '001' then                      // Check DLK, OSLK and PU bits
        IMPLEMENTATION_DEFINED "generate error response";
        return bits(32) UNKNOWN;

    underrun = EDSCR.TXfull == '0' || (Halted() && EDSCR.MA == '1' && EDSCR.ITE == '0');
    value = if underrun then bits(32) UNKNOWN else DTRTX;

    if EDSCR.ERR == '1' then return value;              // Error flag set: no side-effects

    if underrun then
        EDSCR.TXU = '1';  EDSCR.ERR = '1';              // Underrun condition: block side-effects
        return value;                                   // Return UNKNOWN

    EDSCR.TXfull = '0';
    if Halted() && EDSCR.MA == '1' then
        EDSCR.ITE = '0';                                // See comments in EDITR[] (external write)

        if !UsingAArch32() then
            ExecuteA64(0xB8404401<31:0>);               // A64 "LDR W1,[X0],#4"
        else
            ExecuteT32(0xF850<15:0> /*hw1*/, 0x1B04<15:0> /*hw2*/);      // T32 "LDR R1,[R0],#4"
        // If the load aborts, the Data Abort exception is taken and EDSCR.ERR is set to 1
        if EDSCR.ERR == '1' then
            EDSCR.TXfull = bit UNKNOWN;
            DBGDTRTX_EL0 = bits(64) UNKNOWN;
        else
            if !UsingAArch32() then
                ExecuteA64(0xD5130501<31:0>);           // A64 "MSR DBGDTRTX_EL0,X1"
            else
                ExecuteT32(0xEE00<15:0> /*hw1*/, 0x1E15<15:0> /*hw2*/);  // T32 "MSR DBGDTRTXint,R1"
            // "MSR DBGDTRTX_EL0,X1" calls DBGDTR_EL0[] (write) which sets TXfull.
            assert EDSCR.TXfull == '1';
        if !UsingAArch32() then
            X[1, 64] = bits(64) UNKNOWN;
        else
            R[1] = bits(32) UNKNOWN;
        EDSCR.ITE = '1';                                // See comments in EDITR[] (external write)

    return value;

// DBGDTRTX_EL0[] (external write)
// ===============================

DBGDTRTX_EL0[boolean memory_mapped] = bits(32) value
    DTRTX = value;
    return;
```

Library pseudocode for shared/debug/dccanditr/DBGDTR_EL0

```
// DBGDTR_EL0[] (write)
// ====================
// System register writes to DBGDTR_EL0, DBGDTRTX_EL0 (AArch64) and DBGDTRTXint (AArch32)

DBGDTR_EL0[] = bits(N) value_in
    bits(N) value = value_in;
    // For MSR DBGDTRTX_EL0,<Rt>  N=32, value=X[t]<31:0>, X[t]<63:32> is ignored
    // For MSR DBGDTR_EL0,<Xt>    N=64, value=X[t]<63:0>
    assert N IN {32,64};
    if EDSCR.TXfull == '1' then
        value = bits(N) UNKNOWN;
    // On a 64-bit write, implement a half-duplex channel
    if N == 64 then DTRRX = value<63:32>;
    DTRTX = value<31:0>;        // 32-bit or 64-bit write
    EDSCR.TXfull = '1';
    return;

// DBGDTR_EL0[] (read)
// ===================
// System register reads of DBGDTR_EL0, DBGDTRRX_EL0 (AArch64) and DBGDTRRXint (AArch32)

bits(N) DBGDTR_EL0[]
    // For MRS <Rt>,DBGDTRTX_EL0  N=32, X[t]=Zeros(32):result
    // For MRS <Xt>,DBGDTR_EL0    N=64, X[t]=result
    assert N IN {32,64};
    bits(N) result;
    if EDSCR.RXfull == '0' then
        result = bits(N) UNKNOWN;
    else
        // On a 64-bit read, implement a half-duplex channel
        // NOTE: the word order is reversed on reads with regards to writes
        if N == 64 then result<63:32> = DTRTX;
        result<31:0> = DTRRX;
    EDSCR.RXfull = '0';
    return result;
```

Library pseudocode for shared/debug/dccanditr/DTR

```
bits(32) DTRRX;
bits(32) DTRTX;
```

Library pseudocode for shared/debug/dccanditr/EDITR

```
// EDITR[] (external write)
// ========================
// Called on writes to debug register 0x084.

EDITR[boolean memory_mapped] = bits(32) value
    if EDPRSR<6:5,0> != '001' then                      // Check DLK, OSLK and PU bits
        IMPLEMENTATION_DEFINED "generate error response";
        return;

    if EDSCR.ERR == '1' then return;                    // Error flag set: ignore write

    if !Halted() then return;                           // Non-debug state: ignore write

    if EDSCR.ITE == '0' || EDSCR.MA == '1' then
        EDSCR.ITO = '1';  EDSCR.ERR = '1';              // Overrun condition: block write
        return;

    // ITE indicates whether the processor is ready to accept another instruction; the processor
    // may support multiple outstanding instructions. Unlike the "InstrCompl" flag in [v7A] there
    // is no indication that the pipeline is empty (all instructions have completed). In this
    // pseudocode, the assumption is that only one instruction can be executed at a time,
    // meaning ITE acts like "InstrCompl".
    EDSCR.ITE = '0';

    if !UsingAArch32() then
        ExecuteA64(value);
    else
        ExecuteT32(value<15:0>/*hw1*/, value<31:16> /*hw2*/);

    EDSCR.ITE = '1';

    return;
```

Library pseudocode for shared/debug/halting/DCPSInstruction

```
// DCPSInstruction()
// =================
// Operation of the DCPS instruction in Debug state

DCPSInstruction(bits(2) target_el)

    SynchronizeContext();

    bits(2) handle_el;
    case target_el of
        when EL1
            if PSTATE.EL == EL2 || (PSTATE.EL == EL3 && !UsingAArch32()) then
                handle_el = PSTATE.EL;
            elsif EL2Enabled() && HCR_EL2.TGE == '1' then
                UNDEFINED;
            else
                handle_el = EL1;
        when EL2
            if !HaveEL(EL2) then
                UNDEFINED;
            elsif PSTATE.EL == EL3 && !UsingAArch32() then
                handle_el = EL3;
            elsif !IsSecureEL2Enabled() && CurrentSecurityState() == SS_Secure then
                UNDEFINED;
            else
                handle_el = EL2;
        when EL3
            if EDSCR.SDD == '1' || !HaveEL(EL3) then
                UNDEFINED;
            else
                handle_el = EL3;
        otherwise
            Unreachable();

    from_secure = CurrentSecurityState() == SS_Secure;
    if ELUsingAArch32(handle_el) then
        if PSTATE.M == M32_Monitor then SCR.NS = '0';
        assert UsingAArch32();                  // Cannot move from AArch64 to AArch32
        case handle_el of
            when EL1
                AArch32.WriteMode(M32_Svc);
                if HavePANExt() && SCTLR.SPAN == '0' then
                    PSTATE.PAN = '1';
            when EL2  AArch32.WriteMode(M32_Hyp);
            when EL3
                AArch32.WriteMode(M32_Monitor);
                if HavePANExt() then
                    if !from_secure then
                        PSTATE.PAN = '0';
                    elsif SCTLR.SPAN == '0' then
                        PSTATE.PAN = '1';
        if handle_el == EL2 then
            ELR_hyp = bits(32) UNKNOWN;  HSR = bits(32) UNKNOWN;
        else
            LR = bits(32) UNKNOWN;
        SPSR[] = bits(32) UNKNOWN;
        PSTATE.E = SCTLR[].EE;
        DLR = bits(32) UNKNOWN;  DSPSR = bits(32) UNKNOWN;

    else                                        // Targeting AArch64
        from_32 = UsingAArch32();
        if from_32 then AArch64.MaybeZeroRegisterUppers();
        if from_32 && HaveSME() && PSTATE.SM == '1' then
            ResetSVEState();
        else
            MaybeZeroSVEUppers(target_el);
        PSTATE.nRW = '0';  PSTATE.SP = '1';  PSTATE.EL = handle_el;
        if HavePANExt() && ((handle_el == EL1 && SCTLR_EL1.SPAN == '0') ||
                            (handle_el == EL2 && HCR_EL2.E2H == '1' &&
                             HCR_EL2.TGE == '1' && SCTLR_EL2.SPAN == '0')) then
```

```
            PSTATE.PAN = '1';
        ELR[] = bits(64) UNKNOWN;  SPSR[] = bits(64) UNKNOWN;  ESR[] = bits(64) UNKNOWN;
        DLR_EL0 = bits(64) UNKNOWN;  DSPSR_EL0 = bits(64) UNKNOWN;
        if HaveUAOExt() then PSTATE.UAO = '0';
        if HaveMTEExt() then PSTATE.TCO = '1';
        if HaveGCS() then PSTATE.EXLOCK = '0';

    UpdateEDSCRFields();                        // Update EDSCR PE state flags
    sync_errors = HaveIESB() && SCTLR[].IESB == '1';
    if HaveDoubleFaultExt() && !UsingAArch32() then
        sync_errors = (sync_errors ||
                       (EffectiveEA() == '1' && SCR_EL3.NMEA == '1' && PSTATE.EL == EL3));
    // SCTLR[].IESB might be ignored in Debug state.
    if !ConstrainUnpredictableBool(Unpredictable_IESBinDebug) then
        sync_errors = FALSE;
    if sync_errors then
        SynchronizeErrors();
    return;
```

Library pseudocode for shared/debug/halting/DRPSInstruction

```
// DRPSInstruction()
// =================
// Operation of the A64 DRPS and T32 ERET instructions in Debug state

DRPSInstruction()

    SynchronizeContext();

    sync_errors = HaveIESB() && SCTLR[].IESB == '1';
    if HaveDoubleFaultExt() && !UsingAArch32() then
        sync_errors = (sync_errors ||
                       (EffectiveEA() == '1' && SCR_EL3.NMEA == '1' && PSTATE.EL == EL3));
    // SCTLR[].IESB might be ignored in Debug state.
    if !ConstrainUnpredictableBool(Unpredictable_IESBinDebug) then
        sync_errors = FALSE;
    if sync_errors then
        SynchronizeErrors();

    DebugRestorePSR();

    return;
```

Library pseudocode for shared/debug/halting/DebugHalt

```
constant bits(6) DebugHalt_Breakpoint      = '000111';
constant bits(6) DebugHalt_EDBGRQ          = '010011';
constant bits(6) DebugHalt_Step_Normal     = '011011';
constant bits(6) DebugHalt_Step_Exclusive  = '011111';
constant bits(6) DebugHalt_OSUnlockCatch   = '100011';
constant bits(6) DebugHalt_ResetCatch      = '100111';
constant bits(6) DebugHalt_Watchpoint      = '101011';
constant bits(6) DebugHalt_HaltInstruction = '101111';
constant bits(6) DebugHalt_SoftwareAccess  = '110011';
constant bits(6) DebugHalt_ExceptionCatch  = '110111';
constant bits(6) DebugHalt_Step_NoSyndrome = '111011';
```

Library pseudocode for shared/debug/halting/DebugRestorePSR

```
// DebugRestorePSR()
// =================

DebugRestorePSR()
    // PSTATE.{N,Z,C,V,Q,GE,SS,D,A,I,F} are not observable and ignored in Debug state, so
    // behave as if UNKNOWN.
    if UsingAArch32() then
        bits(32) spsr = SPSR[];
        SetPSTATEFromPSR(spsr);
        PSTATE.<N,Z,C,V,Q,GE,SS,A,I,F> = bits(13) UNKNOWN;
        //  In AArch32, all instructions are T32 and unconditional.
        PSTATE.IT = '00000000';  PSTATE.T = '1';        // PSTATE.J is RES0
        DLR = bits(32) UNKNOWN;  DSPSR = bits(32) UNKNOWN;
    else
        bits(64) spsr = SPSR[];
        SetPSTATEFromPSR(spsr);
        PSTATE.<N,Z,C,V,SS,D,A,I,F> = bits(9) UNKNOWN;
        DLR_EL0 = bits(64) UNKNOWN;  DSPSR_EL0 = bits(64) UNKNOWN;
    UpdateEDSCRFields();                                // Update EDSCR PE state flags
```

Library pseudocode for shared/debug/halting/DisableITRAndResumeInstructionPrefetch

```
// DisableITRAndResumeInstructionPrefetch()
// ========================================

DisableITRAndResumeInstructionPrefetch();
```

Library pseudocode for shared/debug/halting/ExecuteA64

```
// ExecuteA64()
// ============
// Execute an A64 instruction in Debug state.

ExecuteA64(bits(32) instr);
```

Library pseudocode for shared/debug/halting/ExecuteT32

```
// ExecuteT32()
// ============
// Execute a T32 instruction in Debug state.

ExecuteT32(bits(16) hw1, bits(16) hw2);
```

Library pseudocode for shared/debug/halting/ExitDebugState

```
// ExitDebugState()
// ================

ExitDebugState()
    assert Halted();
    SynchronizeContext();

    // Although EDSCR.STATUS signals that the PE is restarting, debuggers must use EDPRSR.SDR to
    // detect that the PE has restarted.
    EDSCR.STATUS = '000001';                           // Signal restarting
    // Clear any pending Halting debug events
    if Havev8p8Debug() then
        EDESR<3:0> = '0000';
    else
        EDESR<2:0> = '000';

    bits(64) new_pc;
    bits(64) spsr;

    if UsingAArch32() then
        new_pc = ZeroExtend(DLR, 64);
        if Havev8p9Debug() then
            spsr = DSPSR2 : DSPSR;
        else
            spsr = ZeroExtend(DSPSR, 64);
    else
        new_pc = DLR_EL0;
        spsr = DSPSR_EL0;

    boolean illegal_psr_state = IllegalExceptionReturn(spsr);
    // If this is an illegal return, SetPSTATEFromPSR() will set PSTATE.IL.
    SetPSTATEFromPSR(spsr);                            // Can update privileged bits, even at EL0

    boolean branch_conditional = FALSE;
    if UsingAArch32() then
        if ConstrainUnpredictableBool(Unpredictable_RESTARTALIGNPC) then new_pc<0> = '0';
        // AArch32 branch
        BranchTo(new_pc<31:0>, BranchType_DBGEXIT, branch_conditional);
    else
        // If targeting AArch32 then PC[63:32,1:0] might be set to UNKNOWN.
        if illegal_psr_state && spsr<4> == '1' then
            new_pc<63:32> = bits(32) UNKNOWN;
            new_pc<1:0> = bits(2) UNKNOWN;
        if HaveBRBExt() then
            BRBEDebugStateExit(new_pc);
        // A type of branch that is never predicted
        BranchTo(new_pc, BranchType_DBGEXIT, branch_conditional);

    (EDSCR.STATUS,EDPRSR.SDR) = ('000010','1');        // Atomically signal restarted
    UpdateEDSCRFields();                               // Stop signalling PE state
    DisableITRAndResumeInstructionPrefetch();

    return;
```

Library pseudocode for shared/debug/halting/Halt

```
// Halt()
// ======

Halt(bits(6) reason)
    boolean is_async = FALSE;
    Halt(reason, is_async);

// Halt()
// ======

Halt(bits(6) reason, boolean is_async)

    if HaveTME() && TSTATE.depth > 0 then
        FailTransaction(TMFailure_DBG, FALSE);

    CTI_SignalEvent(CrossTriggerIn_CrossHalt);  // Trigger other cores to halt

    bits(64) preferred_restart_address = ThisInstrAddr(64);
    bits(64) spsr = GetPSRFromPSTATE(DebugState, 64);

    if (HaveBTIExt() && !is_async && !(reason IN {DebugHalt_Step_Normal, DebugHalt_Step_Exclusive,
        DebugHalt_Step_NoSyndrome, DebugHalt_Breakpoint, DebugHalt_HaltInstruction}) &&
        ConstrainUnpredictableBool(Unpredictable_ZEROBTYPE)) then
        spsr<11:10> = '00';

    if UsingAArch32() then
        DLR = preferred_restart_address<31:0>;
        DSPSR = spsr<31:0>;
        if Havev8p9Debug() then
            DSPSR2 = spsr<63:32>;
    else
        DLR_EL0 = preferred_restart_address;
        DSPSR_EL0 = spsr;

    EDSCR.ITE = '1';
    EDSCR.ITO = '0';
    if HaveRME() then
        if PSTATE.EL == EL3 then
            EDSCR.SDD = '0';
        else
            EDSCR.SDD = if ExternalRootInvasiveDebugEnabled() then '0' else '1';
    elsif CurrentSecurityState() == SS_Secure then
        EDSCR.SDD = '0';                        // If entered in Secure state, allow debug
    elsif HaveEL(EL3) then
        EDSCR.SDD = if ExternalSecureInvasiveDebugEnabled() then '0' else '1';
    else
        EDSCR.SDD = '1';                        // Otherwise EDSCR.SDD is RES1
    EDSCR.MA = '0';

    // In Debug state:
    // * PSTATE.{SS,SSBS,D,A,I,F} are not observable and ignored so behave-as-if UNKNOWN.
    // * PSTATE.{N,Z,C,V,Q,GE,E,M,nRW,EL,SP,DIT} are also not observable, but since these
    //     are not changed on exception entry, this function also leaves them unchanged.
    // * PSTATE.{IT,T} are ignored.
    // * PSTATE.IL is ignored and behave-as-if 0.
    // * PSTATE.BTYPE is ignored and behave-as-if 0.
    // * PSTATE.TCO is set 1.
    // * PSTATE.{UAO,PAN} are observable and not changed on entry into Debug state.

    if UsingAArch32() then
        PSTATE.<IT,SS,SSBS,A,I,F,T> = bits(14) UNKNOWN;
    else
        PSTATE.<SS,SSBS,D,A,I,F>    = bits(6)  UNKNOWN;

        PSTATE.TCO = '1';
        PSTATE.BTYPE = '00';
    PSTATE.IL = '0';
    StopInstructionPrefetchAndEnableITR();
    EDSCR.STATUS = reason;                      // Signal entered Debug state
    UpdateEDSCRFields();                        // Update EDSCR PE state flags.
```

```
    if !is_async then EndOfInstruction();

    return;
```

Library pseudocode for shared/debug/halting/HaltOnBreakpointOrWatchpoint

```
// HaltOnBreakpointOrWatchpoint()
// ==============================
// Returns TRUE if the Breakpoint and Watchpoint debug events should be considered for Debug
// state entry, FALSE if they should be considered for a debug exception.

boolean HaltOnBreakpointOrWatchpoint()
    return HaltingAllowed() && EDSCR.HDE == '1' && OSLSR_EL1.OSLK == '0';
```

Library pseudocode for shared/debug/halting/Halted

```
// Halted()
// ========

boolean Halted()
    return !(EDSCR.STATUS IN {'000001', '000010'});                     // Halted
```

Library pseudocode for shared/debug/halting/HaltingAllowed

```
// HaltingAllowed()
// ================
// Returns TRUE if halting is currently allowed, FALSE if halting is prohibited.

boolean HaltingAllowed()
    if Halted() || DoubleLockStatus() then
        return FALSE;
    ss = CurrentSecurityState();
    case ss of
        when SS_NonSecure return ExternalInvasiveDebugEnabled();
        when SS_Secure    return ExternalSecureInvasiveDebugEnabled();
        when SS_Root      return ExternalRootInvasiveDebugEnabled();
        when SS_Realm     return ExternalRealmInvasiveDebugEnabled();
```

Library pseudocode for shared/debug/halting/Restarting

```
// Restarting()
// ============

boolean Restarting()
    return EDSCR.STATUS == '000001';                                    // Restarting
```

Library pseudocode for shared/debug/halting/StopInstructionPrefetchAndEnableITR

```
// StopInstructionPrefetchAndEnableITR()
// =====================================

StopInstructionPrefetchAndEnableITR();
```

Library pseudocode for shared/debug/halting/UpdateEDSCRFields

```
// UpdateEDSCRFields()
// ===================
// Update EDSCR PE state fields

UpdateEDSCRFields()

    if !Halted() then
        EDSCR.EL = '00';
        if HaveRME() then
           // SDD bit.
            EDSCR.SDD = if ExternalRootInvasiveDebugEnabled() then '0' else '1';
            EDSCR.<NSE,NS> = bits(2) UNKNOWN;
        else
            // SDD bit.
            EDSCR.SDD = if ExternalSecureInvasiveDebugEnabled() then '0' else '1';
            EDSCR.NS = bit UNKNOWN;

        EDSCR.RW = '1111';
    else
        EDSCR.EL = PSTATE.EL;
        ss = CurrentSecurityState();
        if HaveRME() then
            case ss of
                when SS_Secure    EDSCR.<NSE,NS> = '00';
                when SS_NonSecure EDSCR.<NSE,NS> = '01';
                when SS_Root      EDSCR.<NSE,NS> = '10';
                when SS_Realm     EDSCR.<NSE,NS> = '11';
        else
            EDSCR.NS = if ss == SS_Secure then '0' else '1';

        bits(4) RW;
        RW<1> = if ELUsingAArch32(EL1) then '0' else '1';
        if PSTATE.EL != EL0 then
            RW<0> = RW<1>;
        else
            RW<0> = if UsingAArch32() then '0' else '1';
        if !HaveEL(EL2) || (HaveEL(EL3) && SCR_GEN[].NS == '0' && !IsSecureEL2Enabled()) then
            RW<2> = RW<1>;
        else
            RW<2> = if ELUsingAArch32(EL2) then '0' else '1';
        if !HaveEL(EL3) then
            RW<3> = RW<2>;
        else
            RW<3> = if ELUsingAArch32(EL3) then '0' else '1';

        // The least-significant bits of EDSCR.RW are UNKNOWN if any higher EL is using AArch32.
        if RW<3> == '0' then RW<2:0> = bits(3) UNKNOWN;
        elsif RW<2> == '0' then RW<1:0> = bits(2) UNKNOWN;
        elsif RW<1> == '0' then RW<0> = bit UNKNOWN;
        EDSCR.RW = RW;
    return;
```

Library pseudocode for shared/debug/haltingevents/CheckExceptionCatch

```
// CheckExceptionCatch()
// =====================
// Check whether an Exception Catch debug event is set on the current Exception level

CheckExceptionCatch(boolean exception_entry)
    // Called after an exception entry or exit, that is, such that the Security state
    // and PSTATE.EL are correct for the exception target. When FEAT_Debugv8p2
    // is not implemented, this function might also be called at any time.
    ss = SecurityStateAtEL(PSTATE.EL);
    integer base;

    case ss of
        when SS_Secure    base = 0;
        when SS_NonSecure base = 4;
        when SS_Realm     base = 16;
        when SS_Root      base = 0;
    if HaltingAllowed() then
        boolean halt;
        if HaveExtendedECDebugEvents() then
            exception_exit = !exception_entry;
            increment = if ss == SS_Realm then 4 else 8;
            ctrl = EDECCR<UInt(PSTATE.EL) + base + increment>:EDECCR<UInt(PSTATE.EL) + base>;
            case ctrl of
                when '00'  halt = FALSE;
                when '01'  halt = TRUE;
                when '10'  halt = (exception_exit == TRUE);
                when '11'  halt = (exception_entry == TRUE);
        else
            halt = (EDECCR<UInt(PSTATE.EL) + base> == '1');

        if halt then
            if Havev8p8Debug() && exception_entry then
                EDESR.EC = '1';
            else
                Halt(DebugHalt_ExceptionCatch);
```

Library pseudocode for shared/debug/haltingevents/CheckHaltingStep

```
// CheckHaltingStep()
// ==================
// Check whether EDESR.SS has been set by Halting Step

CheckHaltingStep(boolean is_async)
    if HaltingAllowed() && EDESR.SS == '1' then
        // The STATUS code depends on how we arrived at the state where EDESR.SS == 1.
        if HaltingStep_DidNotStep() then
            Halt(DebugHalt_Step_NoSyndrome, is_async);
        elsif HaltingStep_SteppedEX() then
            Halt(DebugHalt_Step_Exclusive, is_async);
        else
            Halt(DebugHalt_Step_Normal, is_async);
```

Library pseudocode for shared/debug/haltingevents/CheckOSUnlockCatch

```
// CheckOSUnlockCatch()
// ====================
// Called on unlocking the OS Lock to pend an OS Unlock Catch debug event

CheckOSUnlockCatch()
    if ((HaveDoPD() && CTIDEVCTL.OSUCE == '1') ||
        (!HaveDoPD() && EDECR.OSUCE == '1')) then
        if !Halted() then EDESR.OSUC = '1';
```

Library pseudocode for shared/debug/haltingevents/CheckPendingExceptionCatch

```
// CheckPendingExceptionCatch()
// ============================
// Check whether EDESR.EC has been set by an Exception Catch debug event.

CheckPendingExceptionCatch(boolean is_async)
    if Havev8p8Debug() && HaltingAllowed() && EDESR.EC == '1' then
        Halt(DebugHalt_ExceptionCatch, is_async);
```

Library pseudocode for shared/debug/haltingevents/CheckPendingOSUnlockCatch

```
// CheckPendingOSUnlockCatch()
// ===========================
// Check whether EDESR.OSUC has been set by an OS Unlock Catch debug event

CheckPendingOSUnlockCatch()
    if HaltingAllowed() && EDESR.OSUC == '1' then
        boolean is_async = TRUE;
        Halt(DebugHalt_OSUnlockCatch, is_async);
```

Library pseudocode for shared/debug/haltingevents/CheckPendingResetCatch

```
// CheckPendingResetCatch()
// ========================
// Check whether EDESR.RC has been set by a Reset Catch debug event

CheckPendingResetCatch()
    if HaltingAllowed() && EDESR.RC == '1' then
        boolean is_async = TRUE;
        Halt(DebugHalt_ResetCatch, is_async);
```

Library pseudocode for shared/debug/haltingevents/CheckResetCatch

```
// CheckResetCatch()
// =================
// Called after reset

CheckResetCatch()
    if (HaveDoPD() && CTIDEVCTL.RCE == '1') || (!HaveDoPD() && EDECR.RCE == '1') then
        EDESR.RC = '1';
        // If halting is allowed then halt immediately
        if HaltingAllowed() then Halt(DebugHalt_ResetCatch);
```

Library pseudocode for shared/debug/haltingevents/CheckSoftwareAccessToDebugRegisters

```
// CheckSoftwareAccessToDebugRegisters()
// =====================================
// Check for access to Breakpoint and Watchpoint registers.

CheckSoftwareAccessToDebugRegisters()
    os_lock = (if ELUsingAArch32(EL1) then DBGOSLSR.OSLK else OSLSR_EL1.OSLK);
    if HaltingAllowed() && EDSCR.TDA == '1' && os_lock == '0' then
        Halt(DebugHalt_SoftwareAccess);
```

Library pseudocode for shared/debug/haltingevents/CheckTRBEHalt

```
// CheckTRBEHalt()
// ===============

CheckTRBEHalt()
    if !Havev8p9Debug() || !HaveFeatTRBEExt() then
        return;

    if (HaltingAllowed() && TraceBufferEnabled() &&
        TRBSR_EL1.IRQ == '1' && EDECR.TRBE == '1') then
        Halt(DebugHalt_EDBGRQ);
```

Library pseudocode for shared/debug/haltingevents/ExternalDebugRequest

```
// ExternalDebugRequest()
// ======================

ExternalDebugRequest()
    if HaltingAllowed() then
        boolean is_async = TRUE;
        Halt(DebugHalt_EDBGRQ, is_async);
    // Otherwise the CTI continues to assert the debug request until it is taken.
```

Library pseudocode for shared/debug/haltingevents/HaltingStep_DidNotStep

```
// HaltingStep_DidNotStep()
// ========================
// Returns TRUE if the previously executed instruction was executed in the inactive state, that is,
// if it was not itself stepped.

boolean HaltingStep_DidNotStep();
```

Library pseudocode for shared/debug/haltingevents/HaltingStep_SteppedEX

```
// HaltingStep_SteppedEX()
// =======================
// Returns TRUE if the previously executed instruction was a Load-Exclusive class instruction
// executed in the active-not-pending state.

boolean HaltingStep_SteppedEX();
```

Library pseudocode for shared/debug/haltingevents/RunHaltingStep

```
// RunHaltingStep()
// ================

RunHaltingStep(boolean exception_generated, bits(2) exception_target, boolean syscall,
               boolean reset)
    // "exception_generated" is TRUE if the previous instruction generated a synchronous exception
    // or was cancelled by an asynchronous exception.
    //
    // if "exception_generated" is TRUE then "exception_target" is the target of the exception, and
    // "syscall" is TRUE if the exception is a synchronous exception where the preferred return
    // address is the instruction following that which generated the exception.
    //
    // "reset" is TRUE if exiting reset state into the highest EL.

    if reset then assert !Halted();             // Cannot come out of reset halted
    active = EDECR.SS == '1' && !Halted();

    if active && reset then                     // Coming out of reset with EDECR.SS set
        EDESR.SS = '1';
    elsif active && HaltingAllowed() then
        boolean advance;
        if exception_generated && exception_target == EL3 then
            advance = syscall || ExternalSecureInvasiveDebugEnabled();
        else
            advance = TRUE;
        if advance then EDESR.SS = '1';

    return;
```

Library pseudocode for shared/debug/interrupts/ExternalDebugInterruptsDisabled

```
// ExternalDebugInterruptsDisabled()
// =================================
// Determine whether EDSCR disables interrupts routed to 'target'.

boolean ExternalDebugInterruptsDisabled(bits(2) target)
    boolean int_dis;
    SecurityState ss = SecurityStateAtEL(target);
    if Havev8p4Debug() then
        if EDSCR.INTdis[0] == '1' then
            case ss of
                when SS_NonSecure int_dis = ExternalInvasiveDebugEnabled();
                when SS_Secure    int_dis = ExternalSecureInvasiveDebugEnabled();
                when SS_Realm     int_dis = ExternalRealmInvasiveDebugEnabled();
                when SS_Root      int_dis = ExternalRootInvasiveDebugEnabled();
        else
            int_dis = FALSE;
    else
        case target of
            when EL3
                int_dis = (EDSCR.INTdis == '11' && ExternalSecureInvasiveDebugEnabled());
            when EL2
                int_dis = (EDSCR.INTdis IN {'1x'} && ExternalInvasiveDebugEnabled());
            when EL1
                if ss == SS_Secure then
                    int_dis = (EDSCR.INTdis IN {'1x'} && ExternalSecureInvasiveDebugEnabled());
                else
                    int_dis = (EDSCR.INTdis != '00' && ExternalInvasiveDebugEnabled());
    return int_dis;
```

Library pseudocode for shared/debug/pmu

```
array integer PMUEventAccumulator[0..30];  // Accumulates PMU events for a cycle

array boolean PMULastThresholdValue[0..30];// A record of the threshold result for each
```

Library pseudocode for shared/debug/pmu/CYCLE_COUNTER_ID

```
constant integer CYCLE_COUNTER_ID = 31;
```

Library pseudocode for shared/debug/pmu/CheckForPMUOverflow

```
// CheckForPMUOverflow()
// =====================
// Signal Performance Monitors overflow IRQ and CTI overflow events.
// Called before each instruction is executed.

CheckForPMUOverflow()
    boolean check_cnten  = FALSE;
    boolean check_e      = TRUE;
    boolean check_inten  = TRUE;
    boolean include_lo   = TRUE;
    boolean include_hi   = TRUE;
    boolean exclude_cyc  = FALSE;
    boolean exclude_sync = FALSE;

    boolean pmuirq = PMUOverflowCondition(check_e, check_cnten, check_inten,
                                          include_hi, include_lo,
                                          exclude_cyc, exclude_sync);

    SetInterruptRequestLevel(InterruptID_PMUIRQ,  if pmuirq then Signal_High else Signal_Low);
    CTI_SetEventLevel(CrossTriggerIn_PMUOverflow, if pmuirq then Signal_High else Signal_Low);

    // The request remains set until the condition is cleared.
    // For example, an interrupt handler or cross-triggered event handler clears
    // the overflow status flag by writing to PMOVSCLR_EL0.

    if HavePMUv3p9() && Havev8p9Debug() then
        if pmuirq && HaltingAllowed() && EDECR.PME == '1' then
            Halt(DebugHalt_EDBGRQ);

    if ShouldBRBEFreeze() then
        BRBEFreeze();

    return;
```

Library pseudocode for shared/debug/pmu/CountPMUEvents

```
// CountPMUEvents()
// ================
// Return TRUE if counter "idx" should count its event.
// For the cycle counter, idx == CYCLE_COUNTER_ID (32).
// For the instruction counter, idx == INSTRUCTION_COUNTER_ID (33).

boolean CountPMUEvents(integer idx)
    constant integer num_counters = GetNumEventCounters();
    assert (idx == CYCLE_COUNTER_ID || idx < num_counters ||
            (idx == INSTRUCTION_COUNTER_ID && HavePMUv3ICNTR()));

    boolean debug;
    boolean enabled;
    boolean prohibited;
    boolean filtered;
    boolean frozen;
    boolean resvd_for_el2;
    bit E;

    // Event counting is disabled in Debug state
    debug = Halted();

    // Software can reserve some counters for EL2
    resvd_for_el2 = PMUCounterIsHyp(idx);
    ss = CurrentSecurityState();

    // Main enable controls
    case idx of
        when INSTRUCTION_COUNTER_ID
            assert HaveAArch64();
            enabled = PMCR_EL0.E == '1' && PMCNTENSET_EL0.F0 == '1';
        when CYCLE_COUNTER_ID
            if HaveAArch64() then
                enabled = PMCR_EL0.E == '1' && PMCNTENSET_EL0.C == '1';
            else
                enabled = PMCR.E == '1' && PMCNTENSET.C == '1';
        otherwise
            if resvd_for_el2 then
                E = if HaveAArch64() then MDCR_EL2.HPME else HDCR.HPME;
            else
                E = if HaveAArch64() then PMCR_EL0.E else PMCR.E;

            if HaveAArch64() then
                enabled = E == '1' && PMCNTENSET_EL0<idx> == '1';
            else
                enabled = E == '1' && PMCNTENSET<idx> == '1';

    // Event counting is allowed unless it is prohibited by any rule below
    prohibited = FALSE;

    // Event counting in Secure state is prohibited if all of:
    // * EL3 is implemented
    // * One of the following is true:
    //   - EL3 is using AArch64, MDCR_EL3.SPME == 0, and either:
    //     - FEAT_PMUv3p7 is not implemented
    //     - MDCR_EL3.MPMX == 0
    //   - EL3 is using AArch32 and SDCR.SPME == 0
    // * Executing at EL0 using AArch32 and one of the following is true:
    //     - EL3 is using AArch32 and SDER.SUNIDEN == 0
    //     - EL3 is using AArch64, EL1 is using AArch32, and SDER32_EL3.SUNIDEN == 0
    if HaveEL(EL3) && ss == SS_Secure then
        if !ELUsingAArch32(EL3) then
            prohibited = MDCR_EL3.SPME == '0' && HavePMUv3p7() && MDCR_EL3.MPMX == '0';
        else
            prohibited = SDCR.SPME == '0';

        if prohibited && PSTATE.EL == EL0 then
            if ELUsingAArch32(EL3) then
                prohibited = SDER.SUNIDEN == '0';
            elsif ELUsingAArch32(EL1) then
```

```
            prohibited = SDER32_EL3.SUNIDEN == '0';

    // Event counting at EL3 is prohibited if all of:
    // * FEAT_PMUv3p7 is implemented
    // * EL3 is using AArch64
    // * One of the following is true:
    //   - MDCR_EL3.SPME == 0
    //   - PMNx is not reserved for EL2
    // * MDCR_EL3.MPMX == 1
    if !prohibited && HavePMUv3p7() && PSTATE.EL == EL3 && HaveAArch64() then
        prohibited = MDCR_EL3.MPMX == '1' && (MDCR_EL3.SPME == '0' || !resvd_for_el2);

    // Event counting at EL2 is prohibited if all of:
    // * The HPMD Extension is implemented
    // * PMNx is not reserved for EL2
    // * EL2 is using AArch64 and MDCR_EL2.HPMD == 1 or EL2 is using AArch32 and HDCR.HPMD == 1
    if !prohibited && PSTATE.EL == EL2 && HaveHPMDExt() && !resvd_for_el2 then
        hpmd = if HaveAArch64() then MDCR_EL2.HPMD else HDCR.HPMD;
        prohibited = hpmd == '1';

    // The IMPLEMENTATION DEFINED authentication interface might override software
    if prohibited && !HaveNoSecurePMUDisableOverride() then
        prohibited = !ExternalSecureNoninvasiveDebugEnabled();

    // Event counting might be frozen
    frozen = FALSE;

    // If FEAT_PMUv3p7 is implemented, event counting can be frozen
    if HavePMUv3p7() then
        bit FZ;
        if resvd_for_el2 then
            FZ = if HaveAArch64() then MDCR_EL2.HPMFZO else HDCR.HPMFZO;
        else
            FZ = if HaveAArch64() then PMCR_EL0.FZO else PMCR.FZO;

        frozen = (FZ == '1') && HiLoPMUOverflow(resvd_for_el2);

    // PMCR_EL0.DP or PMCR.DP disables the cycle counter when event counting is prohibited
    if (prohibited || frozen) && idx == CYCLE_COUNTER_ID then
        dp = if HaveAArch64() then PMCR_EL0.DP else PMCR.DP;
        enabled = enabled && dp == '0';
        // Otherwise whether event counting is prohibited does not affect the cycle counter
        prohibited = FALSE;
        frozen = FALSE;

    // Freeze-on-SPE event is not implemented.

    // If FEAT_PMUv3p5 is implemented, cycle counting can be prohibited.
    // This is not overridden by PMCR_EL0.DP.
    if HavePMUv3p5() && idx == CYCLE_COUNTER_ID then
        if HaveEL(EL3) && ss == SS_Secure then
            sccd = if HaveAArch64() then MDCR_EL3.SCCD else SDCR.SCCD;
            if sccd == '1' then
                prohibited = TRUE;

        if PSTATE.EL == EL2 then
            hccd = if HaveAArch64() then MDCR_EL2.HCCD else HDCR.HCCD;
            if hccd == '1' then
                prohibited = TRUE;

    // If FEAT_PMUv3p7 is implemented, cycle counting an be prohibited at EL3.
    // This is not overriden by PMCR_EL0.DP.
    if HavePMUv3p7() && idx == CYCLE_COUNTER_ID then
        if PSTATE.EL == EL3 && HaveAArch64() && MDCR_EL3.MCCD == '1' then
            prohibited = TRUE;

    // Event counting can be filtered by the {P, U, NSK, NSU, NSH, M, SH, RLK, RLU, RLH} bits
    bits(32) filter;
    case idx of
        when INSTRUCTION_COUNTER_ID
```

```
            filter = PMICFILTR_EL0<31:0>;
        when CYCLE_COUNTER_ID
            filter = if HaveAArch64() then PMCCFILTR_EL0<31:0> else PMCCFILTR;
        otherwise
            filter = if HaveAArch64() then PMEVTYPER_EL0[idx]<31:0> else PMEVTYPER[idx];

    P   = filter<31>;
    U   = filter<30>;
    NSK = if HaveEL(EL3) then filter<29> else '0';
    NSU = if HaveEL(EL3) then filter<28> else '0';
    NSH = if HaveEL(EL2) then filter<27> else '0';
    M   = if HaveEL(EL3) && HaveAArch64() then filter<26> else '0';
    SH  = if HaveEL(EL3) && HaveSecureEL2Ext() then filter<24> else '0';
    RLK = if HaveRME() then filter<22> else '0';
    RLU = if HaveRME() then filter<21> else '0';
    RLH = if HaveRME() then filter<20> else '0';

    ss = CurrentSecurityState();
    case PSTATE.EL of
        when EL0
            case ss of
                when SS_NonSecure filtered = U != NSU;
                when SS_Secure    filtered = U == '1';
                when SS_Realm     filtered = U != RLU;
        when EL1
            case ss of
                when SS_NonSecure filtered = P != NSK;
                when SS_Secure    filtered = P == '1';
                when SS_Realm     filtered = P != RLK;
        when EL2
            case ss of
                when SS_NonSecure filtered = NSH == '0';
                when SS_Secure    filtered = NSH == SH;
                when SS_Realm     filtered = NSH == RLH;
        when EL3
            if HaveAArch64() then
                filtered = M != P;
            else
                filtered = P == '1';

    return !debug && enabled && !prohibited && !filtered && !frozen;
```

Library pseudocode for shared/debug/pmu/GetNumEventCounters

```
// GetNumEventCounters()
// =====================
// Returns the number of event counters implemented. This is indicated to software at the
// highest Exception level by PMCR.N in AArch32 state, and PMCR_EL0.N in AArch64 state.

integer GetNumEventCounters()
    return integer IMPLEMENTATION_DEFINED "Number of event counters";
```

Library pseudocode for shared/debug/pmu/HasElapsed64Cycles

```
// HasElapsed64Cycles()
// ====================
// Returns TRUE if 64 cycles have elapsed between the last count, and FALSE otherwise.

boolean HasElapsed64Cycles();
```

Library pseudocode for shared/debug/pmu/HiLoPMUOverflow

```
// HiLoPMUOverflow()
// =================

boolean HiLoPMUOverflow(boolean resvd_for_el2)
    boolean check_cnten  = FALSE;
    boolean check_e      = FALSE;
    boolean check_inten  = FALSE;
    boolean include_lo   = !resvd_for_el2;
    boolean include_hi   = resvd_for_el2;
    boolean exclude_cyc  = FALSE;
    boolean exclude_sync = FALSE;

    boolean overflow = PMUOverflowCondition(check_e, check_cnten, check_inten,
                                            include_hi, include_lo,
                                            exclude_cyc, exclude_sync);
    return overflow;
```

Library pseudocode for shared/debug/pmu/INSTRUCTION_COUNTER_ID

```
constant integer INSTRUCTION_COUNTER_ID = 32;
```

Library pseudocode for shared/debug/pmu/IncrementInstructionCounter

```
// IncrementInstructionCounter()
// =============================
// Increment the instruction counter and possibly set overflow bits.

IncrementInstructionCounter(integer increment)
    if CountPMUEvents(INSTRUCTION_COUNTER_ID) then
        integer old_value = UInt(PMICNTR_EL0);
        integer new_value = old_value + increment;
        PMICNTR_EL0       = new_value<63:0>;

        // The effective value of PMCR_EL0.LP is '1' for the instruction counter
        if old_value<64> != new_value<64> then
            PMOVSSET_EL0.F0 = '1';
            PMOVSCLR_EL0.F0 = '1';
```

Library pseudocode for shared/debug/pmu/PMUCountValue

```
// PMUCountValue()
// ===============
// Implements the PMU threshold function, if implemented.
// Returns the value to increment event counter 'n' by.
// 'Vb' is the base value of the event that event counter 'n' is configured to count.

integer PMUCountValue(integer n, integer Vb)
    if !HavePMUv3TH() || !HaveAArch64() then
        return Vb;

    integer T = UInt(PMEVTYPER_EL0[n].TH);
    boolean Vc;

    case PMEVTYPER_EL0[n].TC<2:1> of
        when '00' Vc = (Vb != T);       // Disabled or not-equal
        when '01' Vc = (Vb == T);       // Equals
        when '10' Vc = (Vb >= T);       // Greater-than-or-equal
        when '11' Vc = (Vb <  T);       // Less-than

    integer Vt;
    if PMEVTYPER_EL0[n].TC<0> == '0' then
        Vt = (if Vc then Vb else 0);    // Count values
    else
        Vt = (if Vc then 1 else 0);     // Count matches

    integer v;
    if HavePMUv3EDGE() && PMEVTYPER_EL0[n].TE == '1' then
        Vp = PMULastThresholdValue[n];

        tc = PMEVTYPER_EL0[n].TC<1:0>;
        // Check for reserved case
        if tc == '00' then
            Constraint c;
            (c, tc) = ConstrainUnpredictableBits(Unpredictable_RESTC, 2);
            if c == Constraint_DISABLED then tc = '00';
            // Otherwise the value returned by ConstrainUnpredictableBits
            // must be a not-reserved value.

        case tc of
            when '00' v = Vt;                           // Reserved - treat as disabled
            when '10' v = (if Vp != Vc  then 1 else 0); // Both edges
            when 'x1' v = (if !Vp && Vc then 1 else 0); // Single edge
    else
        v = Vt;

    PMULastThresholdValue[n] = Vc;

    return v;
```

Library pseudocode for shared/debug/pmu/PMUCounterIsHyp

```
// PMUCounterIsHyp()
// =================
// Returns TRUE if a counter is reserved for use by EL2, FALSE otherwise.

boolean PMUCounterIsHyp(integer n)
    if n == INSTRUCTION_COUNTER_ID then return FALSE;
    if n == CYCLE_COUNTER_ID then return FALSE;

    boolean resvd_for_el2;
    if HaveEL(EL2) then     // Software can reserve some event counters for EL2
        bits(5) hpmn_bits = if HaveAArch64() then MDCR_EL2.HPMN else HDCR.HPMN;
        resvd_for_el2 = n >= UInt(hpmn_bits);
        if UInt(hpmn_bits) > GetNumEventCounters() || (!HaveFeatHPMN0() && IsZero(hpmn_bits)) then
            resvd_for_el2 = ConstrainUnpredictableBool(Unpredictable_CounterReservedForEL2);
    else
        resvd_for_el2 = FALSE;

    return resvd_for_el2;
```

Library pseudocode for shared/debug/pmu/PMUCounterMask

```
// PMUCounterMask()
// ================
// Return bitmask of accessible PMU counters.

bits(64) PMUCounterMask()
    integer n;
    if UsingAArch32() then
        n = AArch32.GetNumEventCountersAccessible();
    else
        n = AArch64.GetNumEventCountersAccessible();

    mask = ZeroExtend(Ones(n), 64);
    mask<CYCLE_COUNTER_ID> = '1';
    if HaveAArch64() && HavePMUv3ICNTR() then mask<INSTRUCTION_COUNTER_ID> = '1';
    return mask;
```

Library pseudocode for shared/debug/pmu/PMUEvent

```
// PMUEvent()
// ==========
// Generate a PMU event. By default, increment by 1.

PMUEvent(bits(16) pmuevent)
    PMUEvent(pmuevent, 1);

// PMUEvent()
// ==========
// Accumulate a PMU Event.

PMUEvent(bits(16) pmuevent, integer increment)
    if SPESampleInFlight then
        SPEEvent(pmuevent);
    integer counters = GetNumEventCounters();
    if counters != 0 then
        for idx = 0 to counters - 1
            PMUEvent(pmuevent, increment, idx);

    if HaveAArch64() && HavePMUv3ICNTR() && pmuevent == PMU_EVENT_INST_RETIRED then
        IncrementInstructionCounter(increment);

// PMUEvent()
// ==========
// Accumulate a PMU Event for a specific event counter.

PMUEvent(bits(16) pmuevent, integer increment, integer idx)
    if !HavePMUv3() then
        return;

    if UsingAArch32() then
        if PMEVTYPER[idx].evtCount == pmuevent then
            PMUEventAccumulator[idx] = PMUEventAccumulator[idx] + increment;
    else
        if PMEVTYPER_EL0[idx].evtCount == pmuevent then
            PMUEventAccumulator[idx] = PMUEventAccumulator[idx] + increment;
```

Library pseudocode for shared/debug/pmu/PMUOverflowCondition

```
// PMUOverflowCondition()
// ======================
// Checks for PMU overflow under certain parameter conditions
// If 'check_e' is TRUE, then check the applicable one of PMCR_EL0.E and MDCR_EL2.HPME.
// If 'check_cnten' is TRUE, then check the applicable PMCNTENCLR_EL0 bit.
// If 'check_cnten' is TRUE, then check the applicable PMINTENCLR_EL1 bit.
// If 'include_lo' is TRUE, then check counters in the set [0..(HPMN-1)], CCNTR
//     and ICNTR, unless excluded by other flags.
// If 'include_hi' is TRUE, then check counters in the set [HPMN..(N-1)].
// If 'exclude_cyc' is TRUE, then CCNTR is NOT checked.
// If 'exclude_sync' is TRUE, then counters in synchronous mode are NOT checked.

boolean PMUOverflowCondition(boolean check_e, boolean check_cnten,
                             boolean check_inten,
                             boolean include_hi, boolean include_lo,
                             boolean exclude_cyc, boolean exclude_sync)
    integer counters = GetNumEventCounters();

    bits(64) ovsf;

    if HaveAArch64() then
        ovsf = PMOVSCLR_EL0;

        // Remove unimplemented counters - these fields are RES0
        ovsf<63:33> = Zeros(31);

        if !HavePMUv3ICNTR() then
            ovsf<INSTRUCTION_COUNTER_ID> = '0';
    else
        ovsf = ZeroExtend(PMOVSR, 64);

    if counters < 31 then
        ovsf<30:counters> = Zeros(31-counters);

    for idx = 0 to counters - 1
        bit E;

        boolean is_hyp = PMUCounterIsHyp(idx);
        if HaveAArch64() then
            E = (if is_hyp then MDCR_EL2.HPME else PMCR_EL0.E);
        else
            E = (if is_hyp then HDCR.HPME else PMCR.E);

        if check_e then
            ovsf<idx> = ovsf<idx> AND E;

        if (!is_hyp && !include_lo) || (is_hyp && !include_hi) then
            ovsf<idx> = '0';

    // Cycle counter
    if exclude_cyc || !include_lo then
        ovsf<CYCLE_COUNTER_ID> = '0';

    if check_e then
        ovsf<CYCLE_COUNTER_ID> = ovsf<CYCLE_COUNTER_ID> AND PMCR_EL0.E;

    // Instruction counter
    if HaveAArch64() && HavePMUv3ICNTR() then
        if !include_lo then
            ovsf<INSTRUCTION_COUNTER_ID> = '0';
        if check_e then
            ovsf<INSTRUCTION_COUNTER_ID> = ovsf<INSTRUCTION_COUNTER_ID> AND PMCR_EL0.E;

    if check_cnten then
        bits(64) cnten = if HaveAArch64() then PMCNTENCLR_EL0 else ZeroExtend(PMCNTENCLR, 64);
        ovsf = ovsf AND cnten;

    if check_inten then
        bits(64) inten = if HaveAArch64() then PMINTENCLR_EL1 else ZeroExtend(PMINTENCLR, 64);
        ovsf = ovsf AND inten;
```

```
    return !IsZero(ovsf);
```

Library pseudocode for shared/debug/samplebasedprofiling/CreatePCSample

```
// CreatePCSample()
// ================

CreatePCSample()
    // In a simple sequential execution of the program, CreatePCSample is executed each time the PE
    // executes an instruction that can be sampled. An implementation is not constrained such that
    // reads of EDPCSRlo return the current values of PC, etc.

    pc_sample.valid = ExternalNoninvasiveDebugAllowed() && !Halted();
    pc_sample.pc = ThisInstrAddr(64);
    pc_sample.el = PSTATE.EL;
    pc_sample.rw = if UsingAArch32() then '0' else '1';
    pc_sample.ss = CurrentSecurityState();
    pc_sample.contextidr = if ELUsingAArch32(EL1) then CONTEXTIDR else CONTEXTIDR_EL1<31:0>;
    pc_sample.has_el2 = PSTATE.EL != EL3 && EL2Enabled();

    if pc_sample.has_el2 then
        if ELUsingAArch32(EL2) then
            pc_sample.vmid = ZeroExtend(VTTBR.VMID, 16);
        elsif !Have16bitVMID() || VTCR_EL2.VS == '0' then
            pc_sample.vmid = ZeroExtend(VTTBR_EL2.VMID<7:0>, 16);
        else
            pc_sample.vmid = VTTBR_EL2.VMID;
        if (HaveVirtHostExt() || HaveV82Debug()) && !ELUsingAArch32(EL2) then
            pc_sample.contextidr_el2 = CONTEXTIDR_EL2<31:0>;
        else
            pc_sample.contextidr_el2 = bits(32) UNKNOWN;
        pc_sample.el0h = PSTATE.EL == EL0 && IsInHost();
    return;
```

Library pseudocode for shared/debug/samplebasedprofiling/EDPCSRlo

```
// EDPCSRlo[] (read)
// =================

bits(32) EDPCSRlo[boolean memory_mapped]

    if EDPRSR<6:5,0> != '001' then                      // Check DLK, OSLK and PU bits
        IMPLEMENTATION_DEFINED "generate error response";
        return bits(32) UNKNOWN;

    // The Software lock is OPTIONAL.
    update = !memory_mapped || EDLSR.SLK == '0';        // Software locked: no side-effects

    bits(32) sample;
    if pc_sample.valid then
        sample = pc_sample.pc<31:0>;
        if update then
            if HaveVirtHostExt() && EDSCR.SC2 == '1' then
                EDPCSRhi.PC = (if pc_sample.rw == '0' then Zeros(24) else pc_sample.pc<55:32>);
                EDPCSRhi.EL = pc_sample.el;
                EDPCSRhi.NS = (if pc_sample.ss == SS_Secure then '0' else '1');
            else
                EDPCSRhi = (if pc_sample.rw == '0' then Zeros(32) else pc_sample.pc<63:32>);
            EDCIDSR = pc_sample.contextidr;
            if (HaveVirtHostExt() || HaveV82Debug()) && EDSCR.SC2 == '1' then
                EDVIDSR = (if pc_sample.has_el2 then pc_sample.contextidr_el2
                           else bits(32) UNKNOWN);
            else
                EDVIDSR.VMID = (if pc_sample.has_el2 && pc_sample.el IN {EL1,EL0}
                                then pc_sample.vmid else Zeros(16));
                EDVIDSR.NS = (if pc_sample.ss == SS_Secure then '0' else '1');
                EDVIDSR.E2 = (if pc_sample.el == EL2 then '1' else '0');
                EDVIDSR.E3 = (if pc_sample.el == EL3 then '1' else '0') AND pc_sample.rw;
                // The conditions for setting HV are not specified if PCSRhi is zero.
                // An example implementation may be "pc_sample.rw".
                EDVIDSR.HV = (if !IsZero(EDPCSRhi) then '1'
                              else bit IMPLEMENTATION_DEFINED "0 or 1");
    else
        sample = Ones(32);
        if update then
            EDPCSRhi = bits(32) UNKNOWN;
            EDCIDSR = bits(32) UNKNOWN;
            EDVIDSR = bits(32) UNKNOWN;

    return sample;
```

Library pseudocode for shared/debug/samplebasedprofiling/PCSample

```
PCSample pc_sample;

// PCSample
// ========

type PCSample is (
    boolean valid,
    bits(64) pc,
    bits(2) el,
    bit rw,
    SecurityState ss,
    boolean has_el2,
    bits(32) contextidr,
    bits(32) contextidr_el2,
    boolean el0h,
    bits(16) vmid
)
```

Library pseudocode for shared/debug/samplebasedprofiling/PMPCSR

```
// PMPCSR[] (read)
// ===============

bits(32) PMPCSR[boolean memory_mapped]
    if EDPRSR<6:5,0> != '001' then                      // Check DLK, OSLK and PU bits
        IMPLEMENTATION_DEFINED "generate error response";
        return bits(32) UNKNOWN;

    // The Software lock is OPTIONAL.
    update = !memory_mapped || PMLSR.SLK == '0';        // Software locked: no side-effects

    bits(32) sample;
    if pc_sample.valid then
        sample = pc_sample.pc<31:0>;
        if update then
            PMPCSR<55:32> = (if pc_sample.rw == '0' then Zeros(24) else pc_sample.pc<55:32>);
            PMPCSR.EL = pc_sample.el;
            if HaveRME() then
                case pc_sample.ss of
                    when SS_Secure
                        PMPCSR.NSE = '0'; PMPCSR.NS = '0';
                    when SS_NonSecure
                        PMPCSR.NSE = '0'; PMPCSR.NS = '1';
                    when SS_Root
                        PMPCSR.NSE = '1'; PMPCSR.NS = '0';
                    when SS_Realm
                        PMPCSR.NSE = '1'; PMPCSR.NS = '1';
            else
                PMPCSR.NS = (if pc_sample.ss == SS_Secure then '0' else '1');

            PMCID1SR = pc_sample.contextidr;
            PMCID2SR = if pc_sample.has_el2 then pc_sample.contextidr_el2 else bits(32) UNKNOWN;

            PMVIDSR.VMID = (if pc_sample.has_el2 && pc_sample.el IN {EL1,EL0} && !pc_sample.el0h
                            then pc_sample.vmid else bits(16) UNKNOWN);
    else
        sample = Ones(32);
        if update then
            PMPCSR<55:32>  = bits(24) UNKNOWN;
            PMPCSR.EL = bits(2) UNKNOWN;
            PMPCSR.NS = bit UNKNOWN;

            PMCID1SR = bits(32) UNKNOWN;
            PMCID2SR = bits(32) UNKNOWN;

            PMVIDSR.VMID = bits(16) UNKNOWN;

    return sample;
```

Library pseudocode for shared/debug/softwarestep/CheckSoftwareStep

```
// CheckSoftwareStep()
// ===================
// Take a Software Step exception if in the active-pending state

CheckSoftwareStep()

    // Other self-hosted debug functions will call AArch32.GenerateDebugExceptions() if called from
    // AArch32 state. However, because Software Step is only active when the debug target Exception
    // level is using AArch64, CheckSoftwareStep only calls AArch64.GenerateDebugExceptions().
    step_enabled = (!ELUsingAArch32(DebugTarget()) && AArch64.GenerateDebugExceptions() &&
                    MDSCR_EL1.SS == '1');
    if step_enabled && PSTATE.SS == '0' then
        AArch64.SoftwareStepException();
```

Library pseudocode for shared/debug/softwarestep/DebugExceptionReturnSS

```
// DebugExceptionReturnSS()
// ========================
// Returns value to write to PSTATE.SS on an exception return or Debug state exit.

bit DebugExceptionReturnSS(bits(N) spsr)
    assert Halted() || Restarting() ||  PSTATE.EL != EL0;

    boolean enabled_at_source;
    if Restarting() then
        enabled_at_source = FALSE;
    elsif UsingAArch32() then
        enabled_at_source = AArch32.GenerateDebugExceptions();
    else
        enabled_at_source = AArch64.GenerateDebugExceptions();

    boolean valid;
    bits(2) dest_el;
    if IllegalExceptionReturn(spsr) then
        dest_el = PSTATE.EL;
    else
        (valid, dest_el) = ELFromSPSR(spsr);  assert valid;

    dest_ss = SecurityStateAtEL(dest_el);
    bit mask;
    boolean enabled_at_dest;
    dest_using_32 = (if dest_el == EL0 then spsr<4> == '1' else ELUsingAArch32(dest_el));
    if dest_using_32 then
        enabled_at_dest = AArch32.GenerateDebugExceptionsFrom(dest_el, dest_ss);
    else
        mask = spsr<9>;
        enabled_at_dest = AArch64.GenerateDebugExceptionsFrom(dest_el, dest_ss, mask);

    ELd = DebugTargetFrom(dest_ss);
    bit SS_bit;
    if !ELUsingAArch32(ELd) && MDSCR_EL1.SS == '1' && !enabled_at_source && enabled_at_dest then
        SS_bit = spsr<21>;
    else
        SS_bit = '0';

    return SS_bit;
```

Library pseudocode for shared/debug/softwarestep/SSAdvance

```
// SSAdvance()
// ===========
// Advance the Software Step state machine.

SSAdvance()

    // A simpler implementation of this function just clears PSTATE.SS to zero regardless of the
    // current Software Step state machine. However, this check is made to illustrate that the
    // processor only needs to consider advancing the state machine from the active-not-pending
    // state.
    target = DebugTarget();
    step_enabled = !ELUsingAArch32(target) && MDSCR_EL1.SS == '1';
    active_not_pending = step_enabled && PSTATE.SS == '1';

    if active_not_pending then PSTATE.SS = '0';

    return;
```

Library pseudocode for shared/debug/softwarestep/SoftwareStep_DidNotStep

```
// SoftwareStep_DidNotStep()
// =========================
// Returns TRUE if the previously executed instruction was executed in the
// inactive state, that is, if it was not itself stepped.
// Might return TRUE or FALSE if the previously executed instruction was an ISB
// or ERET executed in the active-not-pending state, or if another exception
// was taken before the Software Step exception.  Returns FALSE otherwise,
// indicating that the previously executed instruction was executed in the
// active-not-pending state, that is, the instruction was stepped.

boolean SoftwareStep_DidNotStep();
```

Library pseudocode for shared/debug/softwarestep/SoftwareStep_SteppedEX

```
// SoftwareStep_SteppedEX()
// ========================
// Returns a value that describes the previously executed instruction. The
// result is valid only if SoftwareStep_DidNotStep() returns FALSE.
// Might return TRUE or FALSE if the instruction was an AArch32 LDREX or LDAEX
// that failed its condition code test.  Otherwise returns TRUE if the
// instruction was a Load-Exclusive class instruction, and FALSE if the
// instruction was not a Load-Exclusive class instruction.
boolean SoftwareStep_SteppedEX();
```

Library pseudocode for shared/exceptions/exceptions/ConditionSyndrome

```
// ConditionSyndrome()
// ===================
// Return CV and COND fields of instruction syndrome

bits(5) ConditionSyndrome()

    bits(5) syndrome;

    if UsingAArch32() then
        cond = AArch32.CurrentCond();
        if PSTATE.T == '0' then             // A32
            syndrome<4> = '1';
            // A conditional A32 instruction that is known to pass its condition code check
            // can be presented either with COND set to 0xE, the value for unconditional, or
            // the COND value held in the instruction.
            if ConditionHolds(cond) && ConstrainUnpredictableBool(Unpredictable_ESRCONDPASS) then
                syndrome<3:0> = '1110';
            else
                syndrome<3:0> = cond;
        else                                // T32
            // When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether:
            //  * CV set to 0 and COND is set to an UNKNOWN value
            //  * CV set to 1 and COND is set to the condition code for the condition that
            //    applied to the instruction.
            if boolean IMPLEMENTATION_DEFINED "Condition valid for trapped T32" then
                syndrome<4> = '1';
                syndrome<3:0> = cond;
            else
                syndrome<4> = '0';
                syndrome<3:0> = bits(4) UNKNOWN;
    else
        syndrome<4> = '1';
        syndrome<3:0> = '1110';

    return syndrome;
```

Library pseudocode for shared/exceptions/exceptions/Exception

```
// Exception
// =========
// Classes of exception.

enumeration Exception {
        Exception_Uncategorized,        // Uncategorized or unknown reason
        Exception_WFxTrap,              // Trapped WFI or WFE instruction
        Exception_CP15RTTrap,           // Trapped AArch32 MCR or MRC access, coproc=0b111
        Exception_CP15RRTTrap,          // Trapped AArch32 MCRR or MRRC access, coproc=0b1111
        Exception_CP14RTTrap,           // Trapped AArch32 MCR or MRC access, coproc=0b1110
        Exception_CP14DTTrap,           // Trapped AArch32 LDC or STC access, coproc=0b1110
        Exception_CP14RRTTrap,          // Trapped AArch32 MRRC access, coproc=0b1110
        Exception_AdvSIMDFPAccessTrap,  // HCPTR-trapped access to SIMD or FP
        Exception_FPIDTrap,             // Trapped access to SIMD or FP ID register
        Exception_LDST64BTrap,          // Trapped access to ST64BV, ST64BV0, ST64B and LD64B
        // Trapped BXJ instruction not supported in Armv8
        Exception_PACTrap,               // Trapped invalid PAC use
        Exception_IllegalState,          // Illegal Execution state
        Exception_SupervisorCall,        // Supervisor Call
        Exception_HypervisorCall,        // Hypervisor Call
        Exception_MonitorCall,           // Monitor Call or Trapped SMC instruction
        Exception_SystemRegisterTrap,    // Trapped MRS or MSR System register access
        Exception_ERetTrap,              // Trapped invalid ERET use
        Exception_InstructionAbort,      // Instruction Abort or Prefetch Abort
        Exception_PCAlignment,           // PC alignment fault
        Exception_DataAbort,             // Data Abort
        Exception_NV2DataAbort,          // Data abort at EL1 reported as being from EL2
        Exception_PACFail,               // PAC Authentication failure
        Exception_SPAlignment,           // SP alignment fault
        Exception_FPTrappedException,    // IEEE trapped FP exception
        Exception_SError,                // SError interrupt
        Exception_Breakpoint,            // (Hardware) Breakpoint
        Exception_SoftwareStep,          // Software Step
        Exception_Watchpoint,            // Watchpoint
        Exception_NV2Watchpoint,         // Watchpoint at EL1 reported as being from EL2
        Exception_SoftwareBreakpoint,    // Software Breakpoint Instruction
        Exception_VectorCatch,           // AArch32 Vector Catch
        Exception_IRQ,                   // IRQ interrupt
        Exception_SVEAccessTrap,         // HCPTR trapped access to SVE
        Exception_SMEAccessTrap,         // HCPTR trapped access to SME
        Exception_TSTARTAccessTrap,      // Trapped TSTART access
        Exception_GPC,                   // Granule protection check
        Exception_BranchTarget,          // Branch Target Identification
        Exception_MemCpyMemSet,          // Exception from a CPY* or SET* instruction
        Exception_GCSFail,               // GCS Exceptions
        Exception_SystemRegister128Trap, // Trapped MRRS or MSRR System register or SYSP access
        Exception_FIQ};                 // FIQ interrupt
```

Library pseudocode for shared/exceptions/exceptions/ExceptionRecord

```
// ExceptionRecord
// ===============

type ExceptionRecord is (
    Exception   exceptype,           // Exception class
    bits(25)    syndrome,            // Syndrome record
    bits(24)    syndrome2,           // Syndrome record
    FullAddress paddress,            // Physical fault address
    bits(64)    vaddress,            // Virtual fault address
    boolean     ipavalid,            // Validity of Intermediate Physical fault address
    boolean     pavalid,             // Validity of Physical fault address
    bit         NS,                  // Intermediate Physical fault address space
    bits(56)    ipaddress,           // Intermediate Physical fault address
    boolean     trappedsyscallinst)  // Trapped SVC or SMC instruction
```

Library pseudocode for shared/exceptions/exceptions/ExceptionSyndrome

```
// ExceptionSyndrome()
// ===================
// Return a blank exception syndrome record for an exception of the given type.

ExceptionRecord ExceptionSyndrome(Exception exceptype)

    ExceptionRecord r;

    r.exceptype = exceptype;

    // Initialize all other fields
    r.syndrome  = Zeros(25);
    r.syndrome2 = Zeros(24);
    r.vaddress  = Zeros(64);
    r.ipavalid  = FALSE;
    r.NS        = '0';
    r.ipaddress = Zeros(56);
    r.paddress.paspace = PASpace UNKNOWN;
    r.paddress.address = bits(56) UNKNOWN;
    r.trappedsyscallinst = FALSE;
    return r;
```

Library pseudocode for shared/functions/aborts/EncodeLDFSC

```
// EncodeLDFSC()
// =============
// Function that gives the Long-descriptor FSC code for types of Fault

bits(6) EncodeLDFSC(Fault statuscode, integer level)
    bits(6) result;

    // 128-bit descriptors will start from level -2 for 4KB to resolve bits IA[55:51]
    if level == -2 then
        assert Have56BitPAExt();
        case statuscode of
            when Fault_AddressSize          result = '101100';
            when Fault_Translation          result = '101010';
            when Fault_SyncExternalOnWalk   result = '010010';
            when Fault_SyncParityOnWalk     result = '011010'; assert !HaveRASExt();
            when Fault_GPCFOnWalk           result = '100010';
            otherwise                       Unreachable();
        return result;

    if level == -1 then
        assert Have52BitIPAAndPASpaceExt();
        case statuscode of
            when Fault_AddressSize          result = '101001';
            when Fault_Translation          result = '101011';
            when Fault_SyncExternalOnWalk   result = '010011';
            when Fault_SyncParityOnWalk     result = '011011'; assert !HaveRASExt();
            when Fault_GPCFOnWalk           result = '100011';
            otherwise                       Unreachable();

        return result;
    case statuscode of
        when Fault_AddressSize         result = '0000':level<1:0>; assert level IN {0,1,2,3};
        when Fault_AccessFlag          result = '0010':level<1:0>; assert level IN {0,1,2,3};
        when Fault_Permission          result = '0011':level<1:0>; assert level IN {0,1,2,3};
        when Fault_Translation         result = '0001':level<1:0>; assert level IN {0,1,2,3};
        when Fault_SyncExternal        result = '010000';
        when Fault_SyncExternalOnWalk  result = '0101':level<1:0>; assert level IN {0,1,2,3};
        when Fault_SyncParity          result = '011000';
        when Fault_SyncParityOnWalk    result = '0111':level<1:0>; assert level IN {0,1,2,3};
        when Fault_AsyncParity         result = '011001';
        when Fault_AsyncExternal       result = '010001'; assert UsingAArch32();
        when Fault_TagCheck            result = '010001'; assert HaveMTE2Ext();
        when Fault_Alignment           result = '100001';
        when Fault_Debug               result = '100010';
        when Fault_GPCFOnWalk          result = '1001':level<1:0>; assert level IN {0,1,2,3};
        when Fault_GPCFOnOutput        result = '101000';
        when Fault_TLBConflict         result = '110000';
        when Fault_HWUpdateAccessFlag  result = '110001';
        when Fault_Lockdown            result = '110100';  // IMPLEMENTATION DEFINED
        when Fault_Exclusive           result = '110101';  // IMPLEMENTATION DEFINED
        otherwise                      Unreachable();

    return result;
```

Library pseudocode for shared/functions/aborts/IPAValid

```
// IPAValid()
// ==========
// Return TRUE if the IPA is reported for the abort

boolean IPAValid(FaultRecord fault)
    assert fault.statuscode != Fault_None;

    if fault.gpcf.gpf != GPCF_None then
        return fault.secondstage;
    elsif fault.s2fs1walk then
        return fault.statuscode IN {
            Fault_AccessFlag,
            Fault_Permission,
            Fault_Translation,
            Fault_AddressSize
        };
    elsif fault.secondstage then
        return fault.statuscode IN {
            Fault_AccessFlag,
            Fault_Translation,
            Fault_AddressSize
        };
    else
        return FALSE;
```

Library pseudocode for shared/functions/aborts/IsAsyncAbort

```
// IsAsyncAbort()
// ==============
// Returns TRUE if the abort currently being processed is an asynchronous abort, and FALSE
// otherwise.

boolean IsAsyncAbort(Fault statuscode)
    assert statuscode != Fault_None;

    return (statuscode IN {Fault_AsyncExternal, Fault_AsyncParity});

// IsAsyncAbort()
// ==============

boolean IsAsyncAbort(FaultRecord fault)
    return IsAsyncAbort(fault.statuscode);
```

Library pseudocode for shared/functions/aborts/IsDebugException

```
// IsDebugException()
// ==================

boolean IsDebugException(FaultRecord fault)
    assert fault.statuscode != Fault_None;
    return fault.statuscode == Fault_Debug;
```

Library pseudocode for shared/functions/aborts/IsExternalAbort

```
// IsExternalAbort()
// =================
// Returns TRUE if the abort currently being processed is an External abort and FALSE otherwise.

boolean IsExternalAbort(Fault statuscode)
    assert statuscode != Fault_None;

    return (statuscode IN {
        Fault_SyncExternal,
        Fault_SyncParity,
        Fault_SyncExternalOnWalk,
        Fault_SyncParityOnWalk,
        Fault_AsyncExternal,
        Fault_AsyncParity
    });

// IsExternalAbort()
// =================

boolean IsExternalAbort(FaultRecord fault)
    return IsExternalAbort(fault.statuscode) || fault.gpcf.gpf == GPCF_EABT;
```

Library pseudocode for shared/functions/aborts/IsExternalSyncAbort

```
// IsExternalSyncAbort()
// =====================
// Returns TRUE if the abort currently being processed is an external
// synchronous abort and FALSE otherwise.

boolean IsExternalSyncAbort(Fault statuscode)
    assert statuscode != Fault_None;

    return (statuscode IN {
        Fault_SyncExternal,
        Fault_SyncParity,
        Fault_SyncExternalOnWalk,
        Fault_SyncParityOnWalk
    });

// IsExternalSyncAbort()
// =====================

boolean IsExternalSyncAbort(FaultRecord fault)
    return IsExternalSyncAbort(fault.statuscode) || fault.gpcf.gpf == GPCF_EABT;
```

Library pseudocode for shared/functions/aborts/IsFault

```
// IsFault()
// =========
// Return TRUE if a fault is associated with an address descriptor

boolean IsFault(AddressDescriptor addrdesc)
    return addrdesc.fault.statuscode != Fault_None;

// IsFault()
// =========
// Return TRUE if a fault is associated with a memory access.

boolean IsFault(Fault fault)
    return fault != Fault_None;

// IsFault()
// =========
// Return TRUE if a fault is associated with status returned by memory.

boolean IsFault(PhysMemRetStatus retstatus)
    return retstatus.statuscode != Fault_None;
```

Library pseudocode for shared/functions/aborts/IsSErrorInterrupt

```
// IsSErrorInterrupt()
// ===================
// Returns TRUE if the abort currently being processed is an SError interrupt, and FALSE
// otherwise.

boolean IsSErrorInterrupt(Fault statuscode)
    assert statuscode != Fault_None;

    return (statuscode IN {Fault_AsyncExternal, Fault_AsyncParity});

// IsSErrorInterrupt()
// ===================

boolean IsSErrorInterrupt(FaultRecord fault)
    return IsSErrorInterrupt(fault.statuscode);
```

Library pseudocode for shared/functions/aborts/IsSecondStage

```
// IsSecondStage()
// ===============

boolean IsSecondStage(FaultRecord fault)
    assert fault.statuscode != Fault_None;

    return fault.secondstage;
```

Library pseudocode for shared/functions/aborts/LSInstructionSyndrome

```
// LSInstructionSyndrome()
// =======================
// Returns the extended syndrome information for a second stage fault.
//  <10>  - Syndrome valid bit. The syndrome is valid only for certain types of access instruction.
//  <9:8> - Access size.
//  <7>   - Sign extended (for loads).
//  <6:2> - Transfer register.
//  <1>   - Transfer register is 64-bit.
//  <0>   - Instruction has acquire/release semantics.

bits(11) LSInstructionSyndrome();
```

Library pseudocode for shared/functions/aborts/ReportAsGPCException

```
// ReportAsGPCException()
// ======================
// Determine whether the given GPCF is reported as a Granule Protection Check Exception
// rather than a Data or Instruction Abort

boolean ReportAsGPCException(FaultRecord fault)
    assert HaveRME();
    assert fault.statuscode IN {Fault_GPCFOnWalk, Fault_GPCFOnOutput};
    assert fault.gpcf.gpf != GPCF_None;

    case fault.gpcf.gpf of
        when GPCF_Walk        return TRUE;
        when GPCF_AddressSize return TRUE;
        when GPCF_EABT        return TRUE;
        when GPCF_Fail        return SCR_EL3.GPF == '1' && PSTATE.EL != EL3;
```

Library pseudocode for shared/functions/cache/CACHE_OP

```
// CACHE_OP()
// ==========
// Performs Cache maintenance operations as per CacheRecord.

CACHE_OP(CacheRecord cache)
    IMPLEMENTATION_DEFINED;
```

Library pseudocode for shared/functions/cache/CPASAtPAS

```
// CPASAtPAS()
// ===========
// Get cache PA space for given PA space.

CachePASpace CPASAtPAS(PASpace pas)
    case pas of
        when PAS_NonSecure
            return CPAS_NonSecure;
        when PAS_Secure
            return CPAS_Secure;
        when PAS_Root
            return CPAS_Root;
        when PAS_Realm
            return CPAS_Realm;
```

Library pseudocode for shared/functions/cache/CPASAtSecurityState

```
// CPASAtSecurityState()
// =====================
// Get cache PA space for given security state.

CachePASpace CPASAtSecurityState(SecurityState ss)
    case ss of
        when SS_NonSecure
            return CPAS_NonSecure;
        when SS_Secure
            return CPAS_SecureNonSecure;
        when SS_Root
            return CPAS_Any;
        when SS_Realm
            return CPAS_RealmNonSecure;
```

Library pseudocode for shared/functions/cache/CacheRecord

```
// CacheRecord
// ===========
// Details related to a cache operation.

type CacheRecord is (
    AccessType       acctype,          // Access type
    CacheOp          cacheop,          // Cache operation
    CacheOpScope     opscope,          // Cache operation type
    CacheType        cachetype,        // Cache type
    bits(64)         regval,
    FullAddress      paddress,
    bits(64)         vaddress,         // For VA operations
    integer          setnum,           // For SW operations
    integer          waynum,           // For SW operations
    integer          level,            // For SW operations
    Shareability     shareability,
    boolean          translated,
    boolean          is_vmid_valid,    // is vmid valid for current context
    bits(16)         vmid,
    boolean          is_asid_valid,    // is asid valid for current context
    bits(16)         asid,
    SecurityState    security,
    // For cache operations to full cache or by setnum/waynum
    // For operations by address, PA space in paddress
    CachePASpace     cpas
)
```

Library pseudocode for shared/functions/cache/DCInstNeedsTranslation

```
// DCInstNeedsTranslation()
// ========================
// Check whether Data Cache operation needs translation.

boolean DCInstNeedsTranslation(CacheOpScope opscope)
    if opscope == CacheOpScope_PoE then
        return FALSE;

    if opscope == CacheOpScope_PoPA then
        return FALSE;

    if CLIDR_EL1.LoC == '000' then
        return !(boolean IMPLEMENTATION_DEFINED
                 "No fault generated for DC operations if PoC is before any level of cache");

    if CLIDR_EL1.LoUU == '000' && opscope == CacheOpScope_PoU then
        return !(boolean IMPLEMENTATION_DEFINED
                 "No fault generated for DC operations if PoU is before any level of cache");

    return TRUE;
```

Library pseudocode for shared/functions/cache/DecodeSW

```
// DecodeSW()
// ==========
// Decode input value into setnum, waynum and level for SW instructions.

(integer, integer, integer) DecodeSW(bits(64) regval, CacheType cachetype)
    level = UInt(regval[3:1]);
    (setnum, waynum, linesize) = GetCacheInfo(level, cachetype);
    return (setnum, waynum, level);
```

Library pseudocode for shared/functions/cache/GetCacheInfo

```
// GetCacheInfo()
// ==============
// Returns numsets, assosciativity & linesize.

(integer, integer, integer) GetCacheInfo(integer level, CacheType cachetype);
```

Library pseudocode for shared/functions/cache/ICInstNeedsTranslation

```
// ICInstNeedsTranslation()
// ========================
// Check whether Instruction Cache operation needs translation.

boolean ICInstNeedsTranslation(CacheOpScope opscope)
    return boolean IMPLEMENTATION_DEFINED "Instruction Cache needs translation";
```

Library pseudocode for shared/functions/common/ASR

```
// ASR()
// =====

bits(N) ASR(bits(N) x, integer shift)
    assert shift >= 0;
    bits(N) result;
    if shift == 0 then
        result = x;
    else
        (result, -) = ASR_C(x, shift);
    return result;
```

Library pseudocode for shared/functions/common/ASR_C

```
// ASR_C()
// =======

(bits(N), bit) ASR_C(bits(N) x, integer shift)
    assert shift > 0 && shift < 256;
    extended_x = SignExtend(x, shift+N);
    result = extended_x<(shift+N)-1:shift>;
    carry_out = extended_x<shift-1>;
    return (result, carry_out);
```

Library pseudocode for shared/functions/common/Abs

```
// Abs()
// =====

integer Abs(integer x)
    return if x >= 0 then x else -x;

// Abs()
// =====

real Abs(real x)
    return if x >= 0.0 then x else -x;
```

Library pseudocode for shared/functions/common/Align

```
// Align()
// =======

integer Align(integer x, integer y)
    return y * (x DIV y);

// Align()
// =======

bits(N) Align(bits(N) x, integer y)
    return Align(UInt(x), y)<N-1:0>;
```

Library pseudocode for shared/functions/common/BitCount

```
// BitCount()
// ==========

integer BitCount(bits(N) x)
    integer result = 0;
    for i = 0 to N-1
        if x<i> == '1' then
            result = result + 1;
    return result;
```

Library pseudocode for shared/functions/common/CountLeadingSignBits

```
// CountLeadingSignBits()
// ======================

integer CountLeadingSignBits(bits(N) x)
    return CountLeadingZeroBits(x<N-1:1> EOR x<N-2:0>);
```

Library pseudocode for shared/functions/common/CountLeadingZeroBits

```
// CountLeadingZeroBits()
// ======================

integer CountLeadingZeroBits(bits(N) x)
    return N - (HighestSetBit(x) + 1);
```

Library pseudocode for shared/functions/common/Elem

```
// Elem[] - non-assignment form
// ============================

bits(size) Elem[bits(N) vector, integer e, integer size]
    assert e >= 0 && (e+1)*size <= N;
    return vector<(e*size+size)-1 : e*size>;

// Elem[] - assignment form
// ========================

Elem[bits(N) &vector, integer e, integer size] = bits(size) value
    assert e >= 0 && (e+1)*size <= N;
    vector<(e+1)*size-1:e*size> = value;
    return;
```

Library pseudocode for shared/functions/common/Extend

```
// Extend()
// ========

bits(N) Extend(bits(M) x, integer N, boolean unsigned)
    return if unsigned then ZeroExtend(x, N) else SignExtend(x, N);
```

Library pseudocode for shared/functions/common/HighestSetBit

```
// HighestSetBit()
// ===============

integer HighestSetBit(bits(N) x)
    for i = N-1 downto 0
        if x<i> == '1' then return i;
    return -1;
```

Library pseudocode for shared/functions/common/Int

```
// Int()
// =====

integer Int(bits(N) x, boolean unsigned)
    result = if unsigned then UInt(x) else SInt(x);
    return result;
```

Library pseudocode for shared/functions/common/IsAligned

```
// IsAligned()
// ===========

boolean IsAligned(integer x, integer y)
    return x == Align(x, y);

// IsAligned()
// ===========

boolean IsAligned(bits(N) x, integer y)
    return x == Align(x, y);
```

Library pseudocode for shared/functions/common/IsOnes

```
// IsOnes()
// ========

boolean IsOnes(bits(N) x)
    return x == Ones(N);
```

Library pseudocode for shared/functions/common/IsZero

```
// IsZero()
// ========

boolean IsZero(bits(N) x)
    return x == Zeros(N);
```

Library pseudocode for shared/functions/common/IsZeroBit

```
// IsZeroBit()
// ===========

bit IsZeroBit(bits(N) x)
    return if IsZero(x) then '1' else '0';
```

Library pseudocode for shared/functions/common/LSL

```
// LSL()
// =====

bits(N) LSL(bits(N) x, integer shift)
    assert shift >= 0;
    bits(N) result;
    if shift == 0 then
        result = x;
    else
        (result, -) = LSL_C(x, shift);
    return result;
```

Library pseudocode for shared/functions/common/LSL_C

```
// LSL_C()
// =======

(bits(N), bit) LSL_C(bits(N) x, integer shift)
    assert shift > 0 && shift < 256;
    extended_x = x : Zeros(shift);
    result = extended_x<N-1:0>;
    carry_out = extended_x<N>;
    return (result, carry_out);
```

Library pseudocode for shared/functions/common/LSR

```
// LSR()
// =====

bits(N) LSR(bits(N) x, integer shift)
    assert shift >= 0;
    bits(N) result;
    if shift == 0 then
        result = x;
    else
        (result, -) = LSR_C(x, shift);
    return result;
```

Library pseudocode for shared/functions/common/LSR_C

```
// LSR_C()
// =======

(bits(N), bit) LSR_C(bits(N) x, integer shift)
    assert shift > 0 && shift < 256;
    extended_x = ZeroExtend(x, shift+N);
    result = extended_x<(shift+N)-1:shift>;
    carry_out = extended_x<shift-1>;
    return (result, carry_out);
```

Library pseudocode for shared/functions/common/LowestSetBit

```
// LowestSetBit()
// ==============

integer LowestSetBit(bits(N) x)
    for i = 0 to N-1
        if x<i> == '1' then return i;
    return N;
```

Library pseudocode for shared/functions/common/Max

```
// Max()
// =====

integer Max(integer a, integer b)
    return if a >= b then a else b;

// Max()
// =====

real Max(real a, real b)
    return if a >= b then a else b;
```

Library pseudocode for shared/functions/common/Min

```
// Min()
// =====

integer Min(integer a, integer b)
    return if a <= b then a else b;

// Min()
// =====

real Min(real a, real b)
    return if a <= b then a else b;
```

Library pseudocode for shared/functions/common/Ones

```
// Ones()
// ======

bits(N) Ones(integer N)
    return Replicate('1',N);
```

Library pseudocode for shared/functions/common/ROR

```
// ROR()
// =====

bits(N) ROR(bits(N) x, integer shift)
    assert shift >= 0;
    bits(N) result;
    if shift == 0 then
        result = x;
    else
        (result, -) = ROR_C(x, shift);
    return result;
```

Library pseudocode for shared/functions/common/ROR_C

```
// ROR_C()
// =======

(bits(N), bit) ROR_C(bits(N) x, integer shift)
    assert shift != 0 && shift < 256;
    m = shift MOD N;
    result = LSR(x,m) OR LSL(x,N-m);
    carry_out = result<N-1>;
    return (result, carry_out);
```

Library pseudocode for shared/functions/common/RShr

```
// RShr()
// ======
// Shift integer value right with rounding

integer RShr(integer value, integer shift, boolean round)
    assert shift > 0;
    if round then
        return (value + (1 << (shift - 1))) >> shift;
    else
        return value >> shift;
```

Library pseudocode for shared/functions/common/Replicate

```
// Replicate()
// ===========

bits(M*N) Replicate(bits(M) x, integer N);
```

Library pseudocode for shared/functions/common/RoundDown

```
// RoundDown()
// ===========

integer RoundDown(real x);
```

Library pseudocode for shared/functions/common/RoundTowardsZero

```
// RoundTowardsZero()
// ==================

integer RoundTowardsZero(real x)
    return if x == 0.0 then 0 else if x >= 0.0 then RoundDown(x) else RoundUp(x);
```

Library pseudocode for shared/functions/common/RoundUp

```
// RoundUp()
// =========

integer RoundUp(real x);
```

Library pseudocode for shared/functions/common/SInt

```
// SInt()
// ======

integer SInt(bits(N) x)
    result = 0;
    for i = 0 to N-1
        if x<i> == '1' then result = result + 2^i;
    if x<N-1> == '1' then result = result - 2^N;
    return result;
```

Library pseudocode for shared/functions/common/SignExtend

```
// SignExtend()
// ============

bits(N) SignExtend(bits(M) x, integer N)
    assert N >= M;
    return Replicate(x<M-1>, N-M) : x;
```

Library pseudocode for shared/functions/common/Split

```
// Split()
// =======

(bits(M-N), bits(N)) Split(bits(M) value, integer N)
    assert M > N;
    return (value<M-1:N>, value<N-1:0>);
```

Library pseudocode for shared/functions/common/UInt

```
// UInt()
// ======

integer UInt(bits(N) x)
    result = 0;
    for i = 0 to N-1
        if x<i> == '1' then result = result + 2^i;
    return result;
```

Library pseudocode for shared/functions/common/ZeroExtend

```
// ZeroExtend()
// ============

bits(N) ZeroExtend(bits(M) x, integer N)
    assert N >= M;
    return Zeros(N-M) : x;
```

Library pseudocode for shared/functions/common/Zeros

```
// Zeros()
// =======

bits(N) Zeros(integer N)
    return Replicate('0',N);
```

Library pseudocode for shared/functions/counters/AArch32.CheckTimerConditions

```
// AArch32.CheckTimerConditions()
// ==============================
// Checking timer conditions for all A32 timer registers

AArch32.CheckTimerConditions()
    boolean status;
    bits(64) offset;
    offset = Zeros(64);
    assert !HaveAArch64();

    if HaveEL(EL3) then
        if CNTP_CTL_S.ENABLE == '1' then
            status = IsTimerConditionMet(offset, CNTP_CVAL_S,
                                         CNTP_CTL_S.IMASK, InterruptID_CNTPS);
            CNTP_CTL_S.ISTATUS = if status then '1' else '0';

        if CNTP_CTL_NS.ENABLE == '1' then
            status = IsTimerConditionMet(offset, CNTP_CVAL_NS,
                                         CNTP_CTL_NS.IMASK, InterruptID_CNTP);
            CNTP_CTL_NS.ISTATUS = if status then '1' else '0';
    else
        if CNTP_CTL.ENABLE == '1' then
            status = IsTimerConditionMet(offset, CNTP_CVAL,
                                         CNTP_CTL.IMASK, InterruptID_CNTP);
            CNTP_CTL.ISTATUS = if status then '1' else '0';

    if HaveEL(EL2) && CNTHP_CTL.ENABLE == '1' then
        status = IsTimerConditionMet(offset, CNTHP_CVAL,
                                     CNTHP_CTL.IMASK, InterruptID_CNTHP);
        CNTHP_CTL.ISTATUS = if status then '1' else '0';

    if CNTV_CTL_EL0.ENABLE == '1' then
        status = IsTimerConditionMet(CNTVOFF_EL2, CNTV_CVAL_EL0,
                                     CNTV_CTL_EL0.IMASK, InterruptID_CNTV);
        CNTV_CTL_EL0.ISTATUS = if status then '1' else '0';

    return;
```

Library pseudocode for shared/functions/counters/AArch64.CheckTimerConditions

```
// AArch64.CheckTimerConditions()
// ==============================
// Checking timer conditions for all A64 timer registers

AArch64.CheckTimerConditions()
    boolean status;
    bits(64) offset;
    bit imask;
    SecurityState ss = CurrentSecurityState();
    boolean ecv = FALSE;
    if HaveECVExt() then
        ecv = CNTHCTL_EL2.ECV == '1' && SCR_EL3.ECVEn == '1' && EL2Enabled();
    if ecv then
        offset = CNTPOFF_EL2;
    else
        offset = Zeros(64);
    if CNTP_CTL_EL0.ENABLE == '1' then
        imask = CNTP_CTL_EL0.IMASK;
        if HaveRME() && ss IN {SS_Root, SS_Realm} && CNTHCTL_EL2.CNTPMASK == '1' then
            imask = '1';
        status = IsTimerConditionMet(offset, CNTP_CVAL_EL0,
                                     imask, InterruptID_CNTP);
        CNTP_CTL_EL0.ISTATUS = if status then '1' else '0';
    if ((HaveEL(EL3) || (HaveEL(EL2) && !HaveSecureEL2Ext())) &&
       CNTHP_CTL_EL2.ENABLE == '1') then
        status = IsTimerConditionMet(Zeros(64), CNTHP_CVAL_EL2,
                                     CNTHP_CTL_EL2.IMASK, InterruptID_CNTHP);
        CNTHP_CTL_EL2.ISTATUS = if status then '1' else '0';
    if HaveEL(EL2) && HaveSecureEL2Ext() && CNTHPS_CTL_EL2.ENABLE == '1' then
        status = IsTimerConditionMet(Zeros(64), CNTHPS_CVAL_EL2,
                                     CNTHPS_CTL_EL2.IMASK, InterruptID_CNTHPS);
        CNTHPS_CTL_EL2.ISTATUS = if status then '1' else '0';

    if CNTPS_CTL_EL1.ENABLE == '1' then
        status = IsTimerConditionMet(offset, CNTPS_CVAL_EL1,
                                     CNTPS_CTL_EL1.IMASK, InterruptID_CNTPS);
        CNTPS_CTL_EL1.ISTATUS = if status then '1' else '0';

    if CNTV_CTL_EL0.ENABLE == '1' then
        imask = CNTV_CTL_EL0.IMASK;
        if HaveRME() && ss IN {SS_Root, SS_Realm} && CNTHCTL_EL2.CNTVMASK == '1' then
            imask = '1';
        status = IsTimerConditionMet(CNTVOFF_EL2, CNTV_CVAL_EL0,
                                     imask, InterruptID_CNTV);
        CNTV_CTL_EL0.ISTATUS = if status then '1' else '0';

    if ((HaveVirtHostExt() && (HaveEL(EL3) || !HaveSecureEL2Ext())) &&
        CNTHV_CTL_EL2.ENABLE == '1') then
        status = IsTimerConditionMet(Zeros(64), CNTHV_CVAL_EL2,
                                     CNTHV_CTL_EL2.IMASK, InterruptID_CNTHV);
        CNTHV_CTL_EL2.ISTATUS = if status then '1' else '0';

    if ((HaveSecureEL2Ext() && HaveVirtHostExt()) &&
        CNTHVS_CTL_EL2.ENABLE == '1') then
        status = IsTimerConditionMet(Zeros(64), CNTHVS_CVAL_EL2,
                                     CNTHVS_CTL_EL2.IMASK, InterruptID_CNTHVS);
        CNTHVS_CTL_EL2.ISTATUS = if status then '1' else '0';
    return;
```

Library pseudocode for shared/functions/counters/GenericCounterTick

```
// GenericCounterTick()
// ====================
// Increments PhysicalCount value for every clock tick.

GenericCounterTick()
    bits(64) prev_physical_count;
    if CNTCR.EN == '0' then
        if !HaveAArch64() then
            AArch32.CheckTimerConditions();
        else
            AArch64.CheckTimerConditions();
        return;
    prev_physical_count = PhysicalCountInt();
    if HaveCNTSCExt() && CNTCR.SCEN == '1' then
        PhysicalCount = PhysicalCount + ZeroExtend(CNTSCR, 88);
    else
        PhysicalCount<87:24> = PhysicalCount<87:24> + 1;
    if !HaveAArch64() then
        AArch32.CheckTimerConditions();
    else
        AArch64.CheckTimerConditions();
    TestEventCNTP(prev_physical_count, PhysicalCountInt());
    TestEventCNTV(prev_physical_count, PhysicalCountInt());
    return;
```

Library pseudocode for shared/functions/counters/IsTimerConditionMet

```
// IsTimerConditionMet()
// =====================

boolean IsTimerConditionMet(bits(64) offset, bits(64) compare_value,
                            bits(1) imask, InterruptID intid)
    boolean condition_met;
    Signal level;
    condition_met = (UInt(PhysicalCountInt() - offset) -
                                         UInt(compare_value)) >= 0;
    level = if condition_met && imask == '0' then Signal_High else Signal_Low;
    SetInterruptRequestLevel(intid, level);
    return condition_met;
```

Library pseudocode for shared/functions/counters/PhysicalCount

```
bits(88) PhysicalCount;
```

Library pseudocode for shared/functions/counters/SetEventRegister

```
// SetEventRegister()
// ==================
// Sets the Event Register of this PE

SetEventRegister()
    EventRegister = '1';
    return;
```

Library pseudocode for shared/functions/counters/TestEventCNTP

```
// TestEventCNTP()
// ===============
// Generate Event stream from the physical counter

TestEventCNTP(bits(64) prev_physical_count, bits(64) current_physical_count)
    bits(64) offset;
    bits(1) samplebit, previousbit;
    if CNTHCTL_EL2.EVNTEN == '1' then
        n = UInt(CNTHCTL_EL2.EVNTI);
        if HaveECVExt() && CNTHCTL_EL2.EVNTIS == '1' then
            n = n + 8;
        boolean ecv = FALSE;
        if HaveECVExt() then
            ecv = (EL2Enabled() && CNTHCTL_EL2.ECV == '1' &&
                   SCR_EL3.ECVEn == '1');
            offset = if ecv then CNTPOFF_EL2 else Zeros(64);
        samplebit   = (current_physical_count - offset)<n>;
        previousbit = (prev_physical_count - offset)<n>;
        if CNTHCTL_EL2.EVNTDIR == '0' then
            if previousbit == '0' && samplebit == '1' then SetEventRegister();
        else
            if previousbit == '1' && samplebit == '0' then SetEventRegister();
    return;
```

Library pseudocode for shared/functions/counters/TestEventCNTV

```
// TestEventCNTV()
// ===============
// Generate Event stream from the virtual counter

TestEventCNTV(bits(64) prev_physical_count, bits(64) current_physical_count)
    bits(64) offset;
    bits(1) samplebit, previousbit;
    if (!(HaveVirtHostExt() && HCR_EL2.<E2H,TGE> == '11') &&
        CNTKCTL_EL1.EVNTEN == '1') then
        n = UInt(CNTKCTL_EL1.EVNTI);
        if HaveECVExt() && CNTKCTL_EL1.EVNTIS == '1' then
            n = n + 8;
        if HaveEL(EL2) && (!EL2Enabled() || HCR_EL2.<E2H,TGE> != '11') then
            offset = CNTVOFF_EL2;
        else
            offset = Zeros(64);
        samplebit   = (current_physical_count - offset)<n>;
        previousbit = (prev_physical_count - offset)<n>;
        if CNTKCTL_EL1.EVNTDIR == '0' then
            if previousbit == '0' && samplebit == '1' then SetEventRegister();
        else
            if previousbit == '1' && samplebit == '0' then SetEventRegister();
    return;
```

Library pseudocode for shared/functions/crc/BitReverse

```
// BitReverse()
// ============

bits(N) BitReverse(bits(N) data)
    bits(N) result;
    for i = 0 to N-1
        result<(N-i)-1> = data<i>;
    return result;
```

Library pseudocode for shared/functions/crc/HaveCRCExt

```
// HaveCRCExt()
// ============

boolean HaveCRCExt()
    return IsFeatureImplemented(FEAT_CRC32);
```

Library pseudocode for shared/functions/crc/Poly32Mod2

```
// Poly32Mod2()
// ============

// Poly32Mod2 on a bitstring does a polynomial Modulus over {0,1} operation

bits(32) Poly32Mod2(bits(N) data_in, bits(32) poly)
    assert N > 32;
    bits(N) data = data_in;
    for i = N-1 downto 32
        if data<i> == '1' then
            data<i-1:0> = data<i-1:0> EOR (poly:Zeros(i-32));
    return data<31:0>;
```

Library pseudocode for shared/functions/crypto/AESInvMixColumns

```
// AESInvMixColumns()
// ==================
// Transformation in the Inverse Cipher that is the inverse of AESMixColumns.

bits(128) AESInvMixColumns(bits (128) op)
    bits(4*8) in0 = op< 96+:8> : op< 64+:8> : op< 32+:8> : op<  0+:8>;
    bits(4*8) in1 = op<104+:8> : op< 72+:8> : op< 40+:8> : op<  8+:8>;
    bits(4*8) in2 = op<112+:8> : op< 80+:8> : op< 48+:8> : op< 16+:8>;
    bits(4*8) in3 = op<120+:8> : op< 88+:8> : op< 56+:8> : op< 24+:8>;

    bits(4*8) out0;
    bits(4*8) out1;
    bits(4*8) out2;
    bits(4*8) out3;

    for c = 0 to 3
        out0<c*8+:8> =  (FFmul0E(in0<c*8+:8>) EOR FFmul0B(in1<c*8+:8>) EOR FFmul0D(in2<c*8+:8>) EOR
                         FFmul09(in3<c*8+:8>));
        out1<c*8+:8> =  (FFmul09(in0<c*8+:8>) EOR FFmul0E(in1<c*8+:8>) EOR FFmul0B(in2<c*8+:8>) EOR
                         FFmul0D(in3<c*8+:8>));
        out2<c*8+:8> =  (FFmul0D(in0<c*8+:8>) EOR FFmul09(in1<c*8+:8>) EOR FFmul0E(in2<c*8+:8>) EOR
                         FFmul0B(in3<c*8+:8>));
        out3<c*8+:8> =  (FFmul0B(in0<c*8+:8>) EOR FFmul0D(in1<c*8+:8>) EOR FFmul09(in2<c*8+:8>) EOR
                         FFmul0E(in3<c*8+:8>));

    return (
        out3<3*8+:8> : out2<3*8+:8> : out1<3*8+:8> : out0<3*8+:8> :
        out3<2*8+:8> : out2<2*8+:8> : out1<2*8+:8> : out0<2*8+:8> :
        out3<1*8+:8> : out2<1*8+:8> : out1<1*8+:8> : out0<1*8+:8> :
        out3<0*8+:8> : out2<0*8+:8> : out1<0*8+:8> : out0<0*8+:8>
    );
```

Library pseudocode for shared/functions/crypto/AESInvShiftRows

```
// AESInvShiftRows()
// =================
// Transformation in the Inverse Cipher that is inverse of AESShiftRows.

bits(128) AESInvShiftRows(bits(128) op)
    return (
        op< 31: 24> : op< 55: 48> : op< 79: 72> : op<103: 96> :
        op<127:120> : op< 23: 16> : op< 47: 40> : op< 71: 64> :
        op< 95: 88> : op<119:112> : op< 15:  8> : op< 39: 32> :
        op< 63: 56> : op< 87: 80> : op<111:104> : op<  7:  0>
    );
```

Library pseudocode for shared/functions/crypto/AESInvSubBytes

```
// AESInvSubBytes()
// ================
// Transformation in the Inverse Cipher that is the inverse of AESSubBytes.

bits(128) AESInvSubBytes(bits(128) op)
    // Inverse S-box values
    bits(16*16*8) GF2_inv = (
        /*       F E D C B A 9 8 7 6 5 4 3 2 1 0       */
        /*F*/ 0x7d0c2155631469e126d677ba7e042b17<127:0> :
        /*E*/ 0x619953833cbbebc8b0f52aae4d3be0a0<127:0> :
        /*D*/ 0xef9cc9939f7ae52d0d4ab519a97f5160<127:0> :
        /*C*/ 0x5fec8027591012b131c7078833a8dd1f<127:0> :
        /*B*/ 0xf45acd78fec0db9a2079d2c64b3e56fc<127:0> :
        /*A*/ 0x1bbe18aa0e62b76f89c5291d711af147<127:0> :
        /*9*/ 0x6edf751ce837f9e28535ade72274ac96<127:0> :
        /*8*/ 0x73e6b4f0cecff297eadc674f4111913a<127:0> :
        /*7*/ 0x6b8a130103bdafc1020f3fca8f1e2cd0<127:0> :
        /*6*/ 0x0645b3b80558e4f70ad3bc8c00abd890<127:0> :
        /*5*/ 0x849d8da75746155edab9edfd5048706c<127:0> :
        /*4*/ 0x92b6655dcc5ca4d41698688664f6f872<127:0> :
        /*3*/ 0x25d18b6d49a25b76b224d92866a12e08<127:0> :
        /*2*/ 0x4ec3fa420b954cee3d23c2a632947b54<127:0> :
        /*1*/ 0xcbe9dec444438e3487ff2f9b8239e37c<127:0> :
        /*0*/ 0xfbd7f3819ea340bf38a53630d56a0952<127:0>
    );
    bits(128) out;
    for i = 0 to 15
        out<i*8+:8> = GF2_inv<UInt(op<i*8+:8>)*8+:8>;
    return out;
```

Library pseudocode for shared/functions/crypto/AESMixColumns

```
// AESMixColumns()
// ===============
// Transformation in the Cipher that takes all of the columns of the
// State and mixes their data (independently of one another) to
// produce new columns.

bits(128) AESMixColumns(bits (128) op)
    bits(4*8) in0 = op< 96+:8> : op< 64+:8> : op< 32+:8> : op<  0+:8>;
    bits(4*8) in1 = op<104+:8> : op< 72+:8> : op< 40+:8> : op<  8+:8>;
    bits(4*8) in2 = op<112+:8> : op< 80+:8> : op< 48+:8> : op< 16+:8>;
    bits(4*8) in3 = op<120+:8> : op< 88+:8> : op< 56+:8> : op< 24+:8>;

    bits(4*8) out0;
    bits(4*8) out1;
    bits(4*8) out2;
    bits(4*8) out3;

    for c = 0 to 3
        out0<c*8+:8> = (FFmul02(in0<c*8+:8>) EOR FFmul03(in1<c*8+:8>) EOR
                                 in2<c*8+:8>  EOR         in3<c*8+:8>);
        out1<c*8+:8> = (FFmul02(in1<c*8+:8>)  EOR FFmul03(in2<c*8+:8>) EOR
                                in3<c*8+:8>   EOR         in0<c*8+:8>);
        out2<c*8+:8> = (FFmul02(in2<c*8+:8>)  EOR FFmul03(in3<c*8+:8>) EOR
                                in0<c*8+:8>   EOR         in1<c*8+:8>);
        out3<c*8+:8> = (FFmul02(in3<c*8+:8>)  EOR FFmul03(in0<c*8+:8>) EOR
                                in1<c*8+:8>   EOR         in2<c*8+:8>);

    return (
        out3<3*8+:8> : out2<3*8+:8> : out1<3*8+:8> : out0<3*8+:8> :
        out3<2*8+:8> : out2<2*8+:8> : out1<2*8+:8> : out0<2*8+:8> :
        out3<1*8+:8> : out2<1*8+:8> : out1<1*8+:8> : out0<1*8+:8> :
        out3<0*8+:8> : out2<0*8+:8> : out1<0*8+:8> : out0<0*8+:8>
    );
```

Library pseudocode for shared/functions/crypto/AESShiftRows

```
// AESShiftRows()
// ==============
// Transformation in the Cipher that processes the State by cyclically
// shifting the last three rows of the State by different offsets.

bits(128) AESShiftRows(bits(128) op)
    return (
        op< 95: 88> : op< 55: 48> : op< 15:  8> : op<103: 96> :
        op< 63: 56> : op< 23: 16> : op<111:104> : op< 71: 64> :
        op< 31: 24> : op<119:112> : op< 79: 72> : op< 39: 32> :
        op<127:120> : op< 87: 80> : op< 47: 40> : op<  7:  0>
    );
```

Library pseudocode for shared/functions/crypto/AESSubBytes

```
// AESSubBytes()
// =============
// Transformation in the Cipher that processes the State using a nonlinear
// byte substitution table (S-box) that operates on each of the State bytes
// independently.

bits(128) AESSubBytes(bits(128) op)
    // S-box values
    bits(16*16*8) GF2 = (
        /*       F E D C B A 9 8 7 6 5 4 3 2 1 0       */
        /*F*/ 0x16bb54b00f2d99416842e6bf0d89a18c<127:0> :
        /*E*/ 0xdf2855cee9871e9b948ed9691198f8e1<127:0> :
        /*D*/ 0x9e1dc186b95735610ef6034866b53e70<127:0> :
        /*C*/ 0x8a8bbd4b1f74dde8c6b4a61c2e2578ba<127:0> :
        /*B*/ 0x08ae7a65eaf4566ca94ed58d6d37c8e7<127:0> :
        /*A*/ 0x79e4959162acd3c25c2406490a3a32e0<127:0> :
        /*9*/ 0xdb0b5ede14b8ee4688902a22dc4f8160<127:0> :
        /*8*/ 0x73195d643d7ea7c41744975fec130ccd<127:0> :
        /*7*/ 0xd2f3ff1021dab6bcf5389d928f40a351<127:0> :
        /*6*/ 0xa89f3c507f02f94585334d43fbaaefd0<127:0> :
        /*5*/ 0xcf584c4a39becb6a5bb1fc20ed00d153<127:0> :
        /*4*/ 0x842fe329b3d63b52a05a6e1b1a2c8309<127:0> :
        /*3*/ 0x75b227ebe28012079a059618c323c704<127:0> :
        /*2*/ 0x1531d871f1e5a534ccf73f362693fdb7<127:0> :
        /*1*/ 0xc072a49cafa2d4adf04759fa7dc982ca<127:0> :
        /*0*/ 0x76abd7fe2b670130c56f6bf27b777c63<127:0>
    );
    bits(128) out;
    for i = 0 to 15
        out<i*8+:8> = GF2<UInt(op<i*8+:8>)*8+:8>;
    return out;
```

Library pseudocode for shared/functions/crypto/FFmul02

```
// FFmul02()
// =========

bits(8) FFmul02(bits(8) b)
    bits(256*8) FFmul_02 = (
        /*       F E D C B A 9 8 7 6 5 4 3 2 1 0       */
        /*F*/ 0xE5E7E1E3EDEFE9EBF5F7F1F3FDFFF9FB<127:0> :
        /*E*/ 0xC5C7C1C3CDCFC9CBD5D7D1D3DDDFD9DB<127:0> :
        /*D*/ 0xA5A7A1A3ADAFA9ABB5B7B1B3BDBFB9BB<127:0> :
        /*C*/ 0x858781838D8F898B959791939D9F999B<127:0> :
        /*B*/ 0x656761636D6F696B757771737D7F797B<127:0> :
        /*A*/ 0x454741434D4F494B555751535D5F595B<127:0> :
        /*9*/ 0x252721232D2F292B353731333D3F393B<127:0> :
        /*8*/ 0x050701030D0F090B151711131D1F191B<127:0> :
        /*7*/ 0xFEFCFAF8F6F4F2F0EEECEAE8E6E4E2E0<127:0> :
        /*6*/ 0xDEDCDAD8D6D4D2D0CECCCAC8C6C4C2C0<127:0> :
        /*5*/ 0xBEBCBAB8B6B4B2B0AEACAAA8A6A4A2A0<127:0> :
        /*4*/ 0x9E9C9A98969492908E8C8A8886848280<127:0> :
        /*3*/ 0x7E7C7A78767472706E6C6A6866646260<127:0> :
        /*2*/ 0x5E5C5A58565452504E4C4A4846444240<127:0> :
        /*1*/ 0x3E3C3A38363432302E2C2A2826242220<127:0> :
        /*0*/ 0x1E1C1A18161412100E0C0A0806040200<127:0>
    );
    return FFmul_02<UInt(b)*8+:8>;
```

Library pseudocode for shared/functions/crypto/FFmul03

```
// FFmul03()
// =========

bits(8) FFmul03(bits(8) b)
    bits(256*8) FFmul_03 = (
        /*       F E D C B A 9 8 7 6 5 4 3 2 1 0       */
        /*F*/ 0x1A191C1F16151013020104070E0D080B<127:0> :
        /*E*/ 0x2A292C2F2625202332313437 3E3D383B<127:0> :
        /*D*/ 0x7A797C7F767570736261646 76E6D686B<127:0> :
        /*C*/ 0x4A494C4F464540435251545 75E5D585B<127:0> :
        /*B*/ 0xDAD9DCDFD6D5D0D3C2C1C4C7CECDC8CB<127:0> :
        /*A*/ 0xEAE9ECEFE6E5E0E3F2F1F4F7FEFDF8FB<127:0> :
        /*9*/ 0xBAB9BCBFB6B5B0B3A2A1A4A7AEADA8AB<127:0> :
        /*8*/ 0x8A898C8F868580839291949 79E9D989B<127:0> :
        /*7*/ 0x818287848D8E8B88999A9F9C95969390<127:0> :
        /*6*/ 0xB1B2B7B4BDBEBBB8A9AAAFACA5A6A3A0<127:0> :
        /*5*/ 0xE1E2E7E4EDEEEBE8F9FAFFFCF5F6F3F0<127:0> :
        /*4*/ 0xD1D2D7D4DDDEDBD8C9CACFCCC5C6C3C0<127:0> :
        /*3*/ 0x414247444D4E4B48595A5F5C55565350<127:0> :
        /*2*/ 0x717277747D7E7B78696A6F6C65666360<127:0> :
        /*1*/ 0x212227242D2E2B28393A3F3C35363330<127:0> :
        /*0*/ 0x111217141D1E1B18090A0F0C05060300<127:0>
    );
    return FFmul_03<UInt(b)*8+:8>;
```

Library pseudocode for shared/functions/crypto/FFmul09

```
// FFmul09()
// =========

bits(8) FFmul09(bits(8) b)
    bits(256*8) FFmul_09 = (
        /*       F E D C B A 9 8 7 6 5 4 3 2 1 0       */
        /*F*/ 0x464F545D626B70790E071C152A233831<127:0> :
        /*E*/ 0xD6DFC4CDF2FBE0E99E978C85BAB3A8A1<127:0> :
        /*D*/ 0x7D746F6659504B42353C272E1118030A<127:0> :
        /*C*/ 0xEDE4FFF6C9C0DBD2A5ACB7BE8188939A<127:0> :
        /*B*/ 0x3039222B141D060F78716A635C554E47<127:0> :
        /*A*/ 0xA0A9B2BB848D969FE8E1FAF3CCC5DED7<127:0> :
        /*9*/ 0x0B0219102F263D34434A5158676E757C<127:0> :
        /*8*/ 0x9B928980BFB6ADA4D3DAC1C8F7FEE5EC<127:0> :
        /*7*/ 0xAAA3B8B18E879C95E2EBF0F9C6CFD4DD<127:0> :
        /*6*/ 0x3A3328211E170C05727B6069565F444D<127:0> :
        /*5*/ 0x9198838AB5BCA7AED9D0CBC2FDF4EFE6<127:0> :
        /*4*/ 0x0108131A252C373E49405B526D647F76<127:0> :
        /*3*/ 0xDCD5CEC7F8F1EAE3949D868FB0B9A2AB<127:0> :
        /*2*/ 0x4C455E5768617A73040D161F2029323B<127:0> :
        /*1*/ 0xE7EEF5FCC3CAD1D8AFA6BDB48B829990<127:0> :
        /*0*/ 0x777E656C535A41483F362D241B120900<127:0>
    );
    return FFmul_09<UInt(b)*8+:8>;
```

Library pseudocode for shared/functions/crypto/FFmul0B

```
// FFmul0B()
// =========

bits(8) FFmul0B(bits(8) b)
    bits(256*8) FFmul_0B = (
        /*       F E D C B A 9 8 7 6 5 4 3 2 1 0       */
        /*F*/ 0xA3A8B5BE8F849992FBF0EDE6D7DCC1CA<127:0> :
        /*E*/ 0x1318050E3F3429224B405D56676C717A<127:0> :
        /*D*/ 0xD8D3CEC5F4FFE2E9808B969DACA7BAB1<127:0> :
        /*C*/ 0x68637E75444F5259303B262D1C170A01<127:0> :
        /*B*/ 0x555E434879726F640D061B10212A373C<127:0> :
        /*A*/ 0xE5EEF3F8C9C2DFD4BDB6ABA0919A878C<127:0> :
        /*9*/ 0x2E2538330209141F767D606B5A514C47<127:0> :
        /*8*/ 0x9E958883B2B9A4AFC6CDD0DBEAE1FCF7<127:0> :
        /*7*/ 0x545F424978736E650C071A11202B363D<127:0> :
        /*6*/ 0xE4EFF2F9C8C3DED5BCB7AAA1909B868D<127:0> :
        /*5*/ 0x2F2439320308151E777C616A5B504D46<127:0> :
        /*4*/ 0x9F948982B3B8A5AEC7CCD1DAEBE0FDF6<127:0> :
        /*3*/ 0xA2A9B4BF8E859893FAF1ECE7D6DDC0CB<127:0> :
        /*2*/ 0x1219040F3E3528234A415C57666D707B<127:0> :
        /*1*/ 0xD9D2CFC4F5FEE3E8818A979CADA6BBB0<127:0> :
        /*0*/ 0x69627F74454E5358313A272C1D160B00<127:0>
    );
    return FFmul_0B<UInt(b)*8+:8>;
```

Library pseudocode for shared/functions/crypto/FFmul0D

```
// FFmul0D()
// =========

bits(8) FFmul0D(bits(8) b)
    bits(256*8) FFmul_0D = (
        /*       F E D C B A 9 8 7 6 5 4 3 2 1 0       */
        /*F*/ 0x979A8D80A3AEB9B4FFF2E5E8CBC6D1DC<127:0> :
        /*E*/ 0x474A5D50737E69642F2235381B16010C<127:0> :
        /*D*/ 0x2C21363B1815020F44495E53707D6A67<127:0> :
        /*C*/ 0xFCF1E6EBC8C5D2DF94998E83A0ADBAB7<127:0> :
        /*B*/ 0xFAF7E0EDCEC3D4D9929F8885A6ABBCB1<127:0> :
        /*A*/ 0x2A27303D1E130409424F5855767B6C61<127:0> :
        /*9*/ 0x414C5B5675786F622924333E1D10070A<127:0> :
        /*8*/ 0x919C8B86A5A8BFB2F9F4E3EECDC0D7DA<127:0> :
        /*7*/ 0x4D40575A7974636E25283F32111C0B06<127:0> :
        /*6*/ 0x9D90878AA9A4B3BEF5F8EFE2C1CCDBD6<127:0> :
        /*5*/ 0xF6FBECE1C2CFD8D59E938489AAA7B0BD<127:0> :
        /*4*/ 0x262B3C31121F08054E4354597A77606D<127:0> :
        /*3*/ 0x202D3A3714190E034845525F7C71666B<127:0> :
        /*2*/ 0xF0FDEAE7C4C9DED39895828FACA1B6BB<127:0> :
        /*1*/ 0x9B96818CAFA2B5B8F3FEE9E4C7CADDD0<127:0> :
        /*0*/ 0x4B46515C7F726568232E3934171A0D00<127:0>
    );
    return FFmul_0D<UInt(b)*8+:8>;
```

Library pseudocode for shared/functions/crypto/FFmul0E

```
// FFmul0E()
// =========

bits(8) FFmul0E(bits(8) b)
    bits(256*8) FFmul_0E = (
        /*       F E D C B A 9 8 7 6 5 4 3 2 1 0       */
        /*F*/ 0x8D83919FB5BBA9A7FDF3E1EFC5CBD9D7<127:0> :
        /*E*/ 0x6D63717F555B49471D13010F252B3937<127:0> :
        /*D*/ 0x56584A446E60727C26283A341E10020C<127:0> :
        /*C*/ 0xB6B8AAA48E80929CC6C8DAD4FEF0E2EC<127:0> :
        /*B*/ 0x202E3C321816040A505E4C426866747A<127:0> :
        /*A*/ 0xC0CEDCD2F8F6E4EAB0BEACA28886949A<127:0> :
        /*9*/ 0xFBF5E7E9C3CDDFD18B859799B3BDAFA1<127:0> :
        /*8*/ 0x1B150709232D3F316B657779535D4F41<127:0> :
        /*7*/ 0xCCC2D0DEF4FAE8E6BCB2A0AE848A9896<127:0> :
        /*6*/ 0x2C22303E141A08065C52404E646A7876<127:0> :
        /*5*/ 0x17190B052F21333D67697B755F51434D<127:0> :
        /*4*/ 0xF7F9EBE5CFC1D3DD87899B95BFB1A3AD<127:0> :
        /*3*/ 0x616F7D735957454B111F0D032927353B<127:0> :
        /*2*/ 0x818F9D93B9B7A5ABF1FFEDE3C9C7D5DB<127:0> :
        /*1*/ 0xBAB4A6A8828C9E90CAC4D6D8F2FCEEE0<127:0> :
        /*0*/ 0x5A544648626C7E702A243638121C0E00<127:0>
    );
    return FFmul_0E<UInt(b)*8+:8>;
```

Library pseudocode for shared/functions/crypto/HaveAESExt

```
// HaveAESExt()
// ============
// TRUE if AES cryptographic instructions support is implemented,
// FALSE otherwise.

boolean HaveAESExt()
    return IsFeatureImplemented(FEAT_AES);
```

Library pseudocode for shared/functions/crypto/HaveBit128PMULLExt

```
// HaveBit128PMULLExt()
// ====================
// TRUE if 128 bit form of PMULL instructions support is implemented,
// FALSE otherwise.

boolean HaveBit128PMULLExt()
    return IsFeatureImplemented(FEAT_PMULL);
```

Library pseudocode for shared/functions/crypto/HaveSHA1Ext

```
// HaveSHA1Ext()
// =============
// TRUE if SHA1 cryptographic instructions support is implemented,
// FALSE otherwise.

boolean HaveSHA1Ext()
    return IsFeatureImplemented(FEAT_SHA1);
```

Library pseudocode for shared/functions/crypto/HaveSHA256Ext

```
// HaveSHA256Ext()
// ===============
// TRUE if SHA256 cryptographic instructions support is implemented,
// FALSE otherwise.

boolean HaveSHA256Ext()
    return IsFeatureImplemented(FEAT_SHA256);
```

Library pseudocode for shared/functions/crypto/HaveSHA3Ext

```
// HaveSHA3Ext()
// =============
// TRUE if SHA3 cryptographic instructions support is implemented,
// and when SHA1 and SHA2 basic cryptographic instructions support is implemented,
// FALSE otherwise.

boolean HaveSHA3Ext()
    return IsFeatureImplemented(FEAT_SHA3);
```

Library pseudocode for shared/functions/crypto/HaveSHA512Ext

```
// HaveSHA512Ext()
// ===============
// TRUE if SHA512 cryptographic instructions support is implemented,
// and when SHA1 and SHA2 basic cryptographic instructions support is implemented,
// FALSE otherwise.

boolean HaveSHA512Ext()
    return IsFeatureImplemented(FEAT_SHA512);
```

Library pseudocode for shared/functions/crypto/HaveSM3Ext

```
// HaveSM3Ext()
// ============
// TRUE if SM3 cryptographic instructions support is implemented,
// FALSE otherwise.

boolean HaveSM3Ext()
    return IsFeatureImplemented(FEAT_SM3);
```

Library pseudocode for shared/functions/crypto/HaveSM4Ext

```
// HaveSM4Ext()
// ============
// TRUE if SM4 cryptographic instructions support is implemented,
// FALSE otherwise.

boolean HaveSM4Ext()
    return IsFeatureImplemented(FEAT_SM4);
```

Library pseudocode for shared/functions/crypto/ROL

```
// ROL()
// =====

bits(N) ROL(bits(N) x, integer shift)
    assert shift >= 0 && shift <= N;
    if (shift == 0) then
        return x;
    return ROR(x, N-shift);
```

Library pseudocode for shared/functions/crypto/SHA256hash

```
// SHA256hash()
// ============

bits(128) SHA256hash(bits (128) x_in, bits(128) y_in, bits(128) w, boolean part1)
    bits(32) chs, maj, t;
    bits(128) x = x_in;
    bits(128) y = y_in;

    for e = 0 to 3
        chs = SHAchoose(y<31:0>, y<63:32>, y<95:64>);
        maj = SHAmajority(x<31:0>, x<63:32>, x<95:64>);
        t = y<127:96> + SHAhashSIGMA1(y<31:0>) + chs + Elem[w, e, 32];
        x<127:96> = t + x<127:96>;
        y<127:96> = t + SHAhashSIGMA0(x<31:0>) + maj;
        <y, x> = ROL(y : x, 32);
    return (if part1 then x else y);
```

Library pseudocode for shared/functions/crypto/SHAchoose

```
// SHAchoose()
// ===========

bits(32) SHAchoose(bits(32) x, bits(32) y, bits(32) z)
    return (((y EOR z) AND x) EOR z);
```

Library pseudocode for shared/functions/crypto/SHAhashSIGMA0

```
// SHAhashSIGMA0()
// ===============

bits(32) SHAhashSIGMA0(bits(32) x)
    return ROR(x, 2) EOR ROR(x, 13) EOR ROR(x, 22);
```

Library pseudocode for shared/functions/crypto/SHAhashSIGMA1

```
// SHAhashSIGMA1()
// ===============

bits(32) SHAhashSIGMA1(bits(32) x)
    return ROR(x, 6) EOR ROR(x, 11) EOR ROR(x, 25);
```

Library pseudocode for shared/functions/crypto/SHAmajority

```
// SHAmajority()
// =============

bits(32) SHAmajority(bits(32) x, bits(32) y, bits(32) z)
    return ((x AND y) OR ((x OR y) AND z));
```

Library pseudocode for shared/functions/crypto/SHAparity

```
// SHAparity()
// ===========

bits(32) SHAparity(bits(32) x, bits(32) y, bits(32) z)
    return (x EOR y EOR z);
```

Library pseudocode for shared/functions/crypto/Sbox

```
// Sbox()
// ======
// Used in SM4E crypto instruction

bits(8) Sbox(bits(8) sboxin)
    bits(8) sboxout;
    bits(2048) sboxstring = (
        /*       F E D C B A 9 8 7 6 5 4 3 2 1 0       */
        /*F*/ 0xd690e9fecce13db716b614c228fb2c05<127:0> :
        /*E*/ 0x2b679a762abe04c3aa44132649860699<127:0> :
        /*D*/ 0x9c4250f491ef987a33540b43edcfac62<127:0> :
        /*C*/ 0xe4b31ca9c908e89580df94fa758f3fa6<127:0> :
        /*B*/ 0x4707a7fcf37317ba83593c19e6854fa8<127:0> :
        /*A*/ 0x686b81b27164da8bf8eb0f4b70569d35<127:0> :
        /*9*/ 0x1e240e5e6358d1a225227c3b01217887<127:0> :
        /*8*/ 0xd40046579fd327524c3602e7a0c4c89e<127:0> :
        /*7*/ 0xeabf8ad240c738b5a3f7f2cef96115a1<127:0> :
        /*6*/ 0xe0ae5da49b341a55ad933230f58cb1e3<127:0> :
        /*5*/ 0x1df6e22e8266ca60c02923ab0d534e6f<127:0> :
        /*4*/ 0xd5db3745defd8e2f03ff6a726d6c5b51<127:0> :
        /*3*/ 0x8d1baf92bbddbc7f11d95c411f105ad8<127:0> :
        /*2*/ 0x0ac13188a5cd7bbd2d74d012b8e5b4b0<127:0> :
        /*1*/ 0x8969974a0c96777e65b9f109c56ec684<127:0> :
        /*0*/ 0x18f07dec3adc4d2079ee5f3ed7cb3948<127:0>
    );

    sboxout = sboxstring<(255-UInt(sboxin))*8+7:(255-UInt(sboxin))*8>;
    return sboxout;
```

Library pseudocode for shared/functions/exclusive/ClearExclusiveByAddress

```
// ClearExclusiveByAddress()
// =========================
// Clear the global Exclusives monitors for all PEs EXCEPT processorid if they
// record any part of the physical address region of size bytes starting at paddress.
// It is IMPLEMENTATION DEFINED whether the global Exclusives monitor for processorid
// is also cleared if it records any part of the address region.

ClearExclusiveByAddress(FullAddress paddress, integer processorid, integer size);
```

Library pseudocode for shared/functions/exclusive/ClearExclusiveLocal

```
// ClearExclusiveLocal()
// =====================
// Clear the local Exclusives monitor for the specified processorid.

ClearExclusiveLocal(integer processorid);
```

Library pseudocode for shared/functions/exclusive/ClearExclusiveMonitors

```
// ClearExclusiveMonitors()
// ========================
// Clear the local Exclusives monitor for the executing PE.

ClearExclusiveMonitors()
    ClearExclusiveLocal(ProcessorID());
```

Library pseudocode for shared/functions/exclusive/ExclusiveMonitorsStatus

```
// ExclusiveMonitorsStatus()
// =========================
// Returns '0' to indicate success if the last memory write by this PE was to
// the same physical address region endorsed by ExclusiveMonitorsPass().
// Returns '1' to indicate failure if address translation resulted in a different
// physical address.

bit ExclusiveMonitorsStatus();
```

Library pseudocode for shared/functions/exclusive/IsExclusiveGlobal

```
// IsExclusiveGlobal()
// ===================
// Return TRUE if the global Exclusives monitor for processorid includes all of
// the physical address region of size bytes starting at paddress.

boolean IsExclusiveGlobal(FullAddress paddress, integer processorid, integer size);
```

Library pseudocode for shared/functions/exclusive/IsExclusiveLocal

```
// IsExclusiveLocal()
// ==================
// Return TRUE if the local Exclusives monitor for processorid includes all of
// the physical address region of size bytes starting at paddress.

boolean IsExclusiveLocal(FullAddress paddress, integer processorid, integer size);
```

Library pseudocode for shared/functions/exclusive/MarkExclusiveGlobal

```
// MarkExclusiveGlobal()
// =====================
// Record the physical address region of size bytes starting at paddress in
// the global Exclusives monitor for processorid.

MarkExclusiveGlobal(FullAddress paddress, integer processorid, integer size);
```

Library pseudocode for shared/functions/exclusive/MarkExclusiveLocal

```
// MarkExclusiveLocal()
// ====================
// Record the physical address region of size bytes starting at paddress in
// the local Exclusives monitor for processorid.

MarkExclusiveLocal(FullAddress paddress, integer processorid, integer size);
```

Library pseudocode for shared/functions/exclusive/ProcessorID

```
// ProcessorID()
// =============
// Return the ID of the currently executing PE.

integer ProcessorID();
```

Library pseudocode for shared/functions/extension/AArch32.HaveHPDExt

```
// AArch32.HaveHPDExt()
// ====================

boolean AArch32.HaveHPDExt()
    return IsFeatureImplemented(FEAT_AA32HPD);
```

Library pseudocode for shared/functions/extension/AArch64.HaveHPDExt

```
// AArch64.HaveHPDExt()
// ====================

boolean AArch64.HaveHPDExt()
    return IsFeatureImplemented(FEAT_HPDS);
```

Library pseudocode for shared/functions/extension/Have128BitDescriptorExt

```
// Have128BitDescriptorExt()
// =========================
// Returns TRUE if 128-bit Descriptor extension
// support is implemented and FALSE otherwise.

boolean Have128BitDescriptorExt()
    return IsFeatureImplemented(FEAT_D128);
```

Library pseudocode for shared/functions/extension/Have16bitVMID

```
// Have16bitVMID()
// ===============
// Returns TRUE if EL2 and support for a 16-bit VMID are implemented.

boolean Have16bitVMID()
    return IsFeatureImplemented(FEAT_VMID16);
```

Library pseudocode for shared/functions/extension/Have52BitIPAAndPASpaceExt

```
// Have52BitIPAAndPASpaceExt()
// ===========================
// Returns TRUE if 52-bit IPA and PA extension support
// is implemented, and FALSE otherwise.

boolean Have52BitIPAAndPASpaceExt()
    return IsFeatureImplemented(FEAT_LPA2);
```

Library pseudocode for shared/functions/extension/Have52BitPAExt

```
// Have52BitPAExt()
// ================
// Returns TRUE if Large Physical Address extension
// support is implemented and FALSE otherwise.

boolean Have52BitPAExt()
    return IsFeatureImplemented(FEAT_LPA);
```

Library pseudocode for shared/functions/extension/Have52BitVAExt

```
// Have52BitVAExt()
// ================
// Returns TRUE if Large Virtual Address extension
// support is implemented and FALSE otherwise.

boolean Have52BitVAExt()
    return IsFeatureImplemented(FEAT_LVA);
```

Library pseudocode for shared/functions/extension/Have56BitPAExt

```
// Have56BitPAExt()
// ================
// Returns TRUE if 56-bit Physical Address extension
// support is implemented and FALSE otherwise.

boolean Have56BitPAExt()
    return IsFeatureImplemented(FEAT_D128);
```

Library pseudocode for shared/functions/extension/Have56BitVAExt

```
// Have56BitVAExt()
// ================
// Returns TRUE if 56-bit Virtual Address extension
// support is implemented and FALSE otherwise.

boolean Have56BitVAExt()
    return IsFeatureImplemented(FEAT_LVA3);
```

Library pseudocode for shared/functions/extension/HaveAArch32BF16Ext

```
// HaveAArch32BF16Ext()
// ====================
// Returns TRUE if AArch32 BFloat16 instruction support is implemented, and FALSE otherwise.

boolean HaveAArch32BF16Ext()
    return IsFeatureImplemented(FEAT_AA32BF16);
```

Library pseudocode for shared/functions/extension/HaveAArch32Int8MatMulExt

```
// HaveAArch32Int8MatMulExt()
// ==========================
// Returns TRUE if AArch32 8-bit integer matrix multiply instruction support
// implemented, and FALSE otherwise.

boolean HaveAArch32Int8MatMulExt()
    return IsFeatureImplemented(FEAT_AA32I8MM);
```

Library pseudocode for shared/functions/extension/HaveAIEExt

```
// HaveAIEExt()
// ============
// Returns TRUE if AIE extension
// support is implemented and FALSE otherwise.

boolean HaveAIEExt()
    return IsFeatureImplemented(FEAT_AIE);
```

Library pseudocode for shared/functions/extension/HaveAccessFlagUpdateExt

```
// HaveAccessFlagUpdateExt()
// =========================

boolean HaveAccessFlagUpdateExt()
    return IsFeatureImplemented(FEAT_HAFDBS);
```

Library pseudocode for shared/functions/extension/HaveAccessFlagUpdateForTableExt

```
// HaveAccessFlagUpdateForTableExt()
// =================================
// Returns TRUE if support for Access Flag Update for Table Descriptors
// is implemented, and FALSE otherwise.

boolean HaveAccessFlagUpdateForTableExt()
    return IsFeatureImplemented(FEAT_HAFT);
```

Library pseudocode for shared/functions/extension/HaveAltFP

```
// HaveAltFP()
// ===========
// Returns TRUE if alternative Floating-point extension support
// is implemented, and FALSE otherwise.

boolean HaveAltFP()
    return IsFeatureImplemented(FEAT_AFP);
```

Library pseudocode for shared/functions/extension/HaveAtomicExt

```
// HaveAtomicExt()
// ===============

boolean HaveAtomicExt()
    return IsFeatureImplemented(FEAT_LSE);
```

Library pseudocode for shared/functions/extension/HaveBF16Ext

```
// HaveBF16Ext()
// =============
// Returns TRUE if AArch64 BFloat16 instruction support is implemented, and FALSE otherwise.

boolean HaveBF16Ext()
    return IsFeatureImplemented(FEAT_BF16);
```

Library pseudocode for shared/functions/extension/HaveBRBEv1p1

```
// HaveBRBEv1p1()
// ==============
// Returns TRUE if BRBEv1p1 extension is implemented, and FALSE otherwise.

boolean HaveBRBEv1p1()
    return IsFeatureImplemented(FEAT_BRBEv1p1);
```

Library pseudocode for shared/functions/extension/HaveBRBExt

```
// HaveBRBExt()
// ============
// Returns TRUE if Branch Record Buffer Extension is implemented, and FALSE otherwise.

boolean HaveBRBExt()
    return IsFeatureImplemented(FEAT_BRBE);
```

Library pseudocode for shared/functions/extension/HaveBTIExt

```
// HaveBTIExt()
// ============
// Returns TRUE if support for Branch Target Indentification is implemented.

boolean HaveBTIExt()
    return IsFeatureImplemented(FEAT_BTI);
```

Library pseudocode for shared/functions/extension/HaveBlockBBM

```
// HaveBlockBBM()
// ==============
// Returns TRUE if support for changing block size without requiring
// break-before-make is implemented.

boolean HaveBlockBBM()
    return IsFeatureImplemented(FEAT_BBM);
```

Library pseudocode for shared/functions/extension/HaveCNTSCExt

```
// HaveCNTSCExt()
// ==============
// Returns TRUE if the Generic Counter Scaling is implemented, and FALSE
// otherwise.

boolean HaveCNTSCExt()
    return IsFeatureImplemented(FEAT_CNTSC);
```

Library pseudocode for shared/functions/extension/HaveCSSC

```
// HaveCSSC()
// ==========
// Returns TRUE if the Common Short Sequence Compression instructions extension is implemented,
// and FALSE otherwise.

boolean HaveCSSC()
    return IsFeatureImplemented(FEAT_CSSC);
```

Library pseudocode for shared/functions/extension/HaveCommonNotPrivateTransExt

```
// HaveCommonNotPrivateTransExt()
// ==============================

boolean HaveCommonNotPrivateTransExt()
    return IsFeatureImplemented(FEAT_TTCNP);
```

Library pseudocode for shared/functions/extension/HaveDGHExt

```
// HaveDGHExt()
// ============
// Returns TRUE if Data Gathering Hint instruction support is implemented, and
// FALSE otherwise.

boolean HaveDGHExt()
    return IsFeatureImplemented(FEAT_DGH);
```

Library pseudocode for shared/functions/extension/HaveDITExt

```
// HaveDITExt()
// ============

boolean HaveDITExt()
    return IsFeatureImplemented(FEAT_DIT);
```

Library pseudocode for shared/functions/extension/HaveDOTPExt

```
// HaveDOTPExt()
// =============
// Returns TRUE if Dot Product feature support is implemented, and FALSE otherwise.

boolean HaveDOTPExt()
    return IsFeatureImplemented(FEAT_DotProd);
```

Library pseudocode for shared/functions/extension/HaveDirtyBitModifierExt

```
// HaveDirtyBitModifierExt()
// =========================

boolean HaveDirtyBitModifierExt()
    return IsFeatureImplemented(FEAT_HAFDBS);
```

Library pseudocode for shared/functions/extension/HaveDoPD

```
// HaveDoPD()
// ==========
// Returns TRUE if Debug Over Power Down extension
// support is implemented and FALSE otherwise.

boolean HaveDoPD()
    return IsFeatureImplemented(FEAT_DoPD);
```

Library pseudocode for shared/functions/extension/HaveDoubleFault2Ext

```
// HaveDoubleFault2Ext()
// =====================
// Returns TRUE if support for the DoubleFault2 feature is implemented, and FALSE otherwise.

boolean HaveDoubleFault2Ext()
    return IsFeatureImplemented(FEAT_DoubleFault2);
```

Library pseudocode for shared/functions/extension/HaveDoubleFaultExt

```
// HaveDoubleFaultExt()
// ====================

boolean HaveDoubleFaultExt()
    return IsFeatureImplemented(FEAT_DoubleFault);
```

Library pseudocode for shared/functions/extension/HaveDoubleLock

```
// HaveDoubleLock()
// ================
// Returns TRUE if support for the OS Double Lock is implemented.

boolean HaveDoubleLock()
    return IsFeatureImplemented(FEAT_DoubleLock);
```

Library pseudocode for shared/functions/extension/HaveE0PDExt

```
// HaveE0PDExt()
// =============
// Returns TRUE if support for constant fault times for unprivileged accesses
// to the memory map is implemented.

boolean HaveE0PDExt()
    return IsFeatureImplemented(FEAT_E0PD);
```

Library pseudocode for shared/functions/extension/HaveEBF16

```
// HaveEBF16()
// ===========
// Returns TRUE if the EBF16 extension is implemented, FALSE otherwise.

boolean HaveEBF16()
    return IsFeatureImplemented(FEAT_EBF16);
```

Library pseudocode for shared/functions/extension/HaveECVExt

```
// HaveECVExt()
// ============
// Returns TRUE if Enhanced Counter Virtualization extension
// support is implemented, and FALSE otherwise.

boolean HaveECVExt()
    return IsFeatureImplemented(FEAT_ECV);
```

Library pseudocode for shared/functions/extension/HaveETExt

```
// HaveETExt()
// ===========
// Returns TRUE if Embedded Trace Extension is implemented, and FALSE otherwise.

boolean HaveETExt()
    return IsFeatureImplemented(FEAT_ETE);
```

Library pseudocode for shared/functions/extension/HaveExtendedCacheSets

```
// HaveExtendedCacheSets()
// =======================

boolean HaveExtendedCacheSets()
    return IsFeatureImplemented(FEAT_CCIDX);
```

Library pseudocode for shared/functions/extension/HaveExtendedECDebugEvents

```
// HaveExtendedECDebugEvents()
// ===========================

boolean HaveExtendedECDebugEvents()
    return IsFeatureImplemented(FEAT_Debugv8p2);
```

Library pseudocode for shared/functions/extension/HaveExtendedExecuteNeverExt

```
// HaveExtendedExecuteNeverExt()
// =============================

boolean HaveExtendedExecuteNeverExt()
    return IsFeatureImplemented(FEAT_XNX);
```

Library pseudocode for shared/functions/extension/HaveFCADDExt

```
// HaveFCADDExt()
// ==============

boolean HaveFCADDExt()
    return IsFeatureImplemented(FEAT_FCMA);
```

Library pseudocode for shared/functions/extension/HaveFGTExt

```
// HaveFGTExt()
// ============
// Returns TRUE if Fine-Grained Traps is implemented, and FALSE otherwise.

boolean HaveFGTExt()
    return IsFeatureImplemented(FEAT_FGT);
```

Library pseudocode for shared/functions/extension/HaveFJCVTZSExt

```
// HaveFJCVTZSExt()
// ================

boolean HaveFJCVTZSExt()
    return IsFeatureImplemented(FEAT_JSCVT);
```

Library pseudocode for shared/functions/extension/HaveFP16MulNoRoundingToFP32Ext

```
// HaveFP16MulNoRoundingToFP32Ext()
// ================================
// Returns TRUE if has FP16 multiply with no intermediate rounding accumulate
// to FP32 instructions, and FALSE otherwise

boolean HaveFP16MulNoRoundingToFP32Ext()
    return IsFeatureImplemented(FEAT_FHM);
```

Library pseudocode for shared/functions/extension/HaveFeatABLE

```
// HaveFeatABLE()
// ==============
// Returns TRUE if support for linking watchpoints to address matching
// breakpoints is implemented, and FALSE otherwise.

boolean HaveFeatABLE()
    return IsFeatureImplemented(FEAT_ABLE);
```

Library pseudocode for shared/functions/extension/HaveFeatCLRBHB

```
// HaveFeatCLRBHB()
// ================
// Returns TRUE if the CLRBHB instruction is implemented, and FALSE otherwise.

boolean HaveFeatCLRBHB()
    return IsFeatureImplemented(FEAT_CLRBHB);
```

Library pseudocode for shared/functions/extension/HaveFeatCMOW

```
// HaveFeatCMOW()
// ==============
// Returns TRUE if the SCTLR_EL1.CMOW bit is implemented and the SCTLR_EL2.CMOW and
// HCRX_EL2.CMOW bits are implemented if EL2 is implemented.

boolean HaveFeatCMOW()
    return IsFeatureImplemented(FEAT_CMOW);
```

Library pseudocode for shared/functions/extension/HaveFeatEBEP

```
// HaveFeatEBEP()
// ==============
// Returns TRUE if the PMU exception is implemented, and FALSE otherwise.

boolean HaveFeatEBEP()
    return IsFeatureImplemented(FEAT_EBEP);
```

Library pseudocode for shared/functions/extension/HaveFeatHBC

```
// HaveFeatHBC()
// =============
// Returns TRUE if the BC instruction is implemented, and FALSE otherwise.

boolean HaveFeatHBC()
    return IsFeatureImplemented(FEAT_HBC);
```

Library pseudocode for shared/functions/extension/HaveFeatHCX

```
// HaveFeatHCX()
// =============
// Returns TRUE if HCRX_EL2 Trap Control register is implemented,
// and FALSE otherwise.

boolean HaveFeatHCX()
    return IsFeatureImplemented(FEAT_HCX);
```

Library pseudocode for shared/functions/extension/HaveFeatHPMN0

```
// HaveFeatHPMN0()
// ===============
// Returns TRUE if HDCR.HPMN or MDCR_EL2.HPMN is permitted to be 0 without
// generating UNPREDICTABLE behavior, and FALSE otherwise.

boolean HaveFeatHPMN0()
    return IsFeatureImplemented(FEAT_HPMN0);
```

Library pseudocode for shared/functions/extension/HaveFeatLS64

```
// HaveFeatLS64()
// ==============
// Returns TRUE if the LD64B, ST64B instructions are
// supported, and FALSE otherwise.

boolean HaveFeatLS64()
    return IsFeatureImplemented(FEAT_LS64);
```

Library pseudocode for shared/functions/extension/HaveFeatLS64_ACCDATA

```
// HaveFeatLS64_ACCDATA()
// ======================
// Returns TRUE if the ST64BV0 instruction is
// supported, and FALSE otherwise.

boolean HaveFeatLS64_ACCDATA()
    return IsFeatureImplemented(FEAT_LS64_ACCDATA);
```

Library pseudocode for shared/functions/extension/HaveFeatLS64_V

```
// HaveFeatLS64_V()
// ================
// Returns TRUE if the ST64BV instruction is
// supported, and FALSE otherwise.

boolean HaveFeatLS64_V()
    return IsFeatureImplemented(FEAT_LS64_V);
```

Library pseudocode for shared/functions/extension/HaveFeatMEC

```
// HaveFeatMEC()
// =============
// Returns TRUE if Memory Encryption Contexts are implemented, and FALSE otherwise.

boolean HaveFeatMEC()
    return IsFeatureImplemented(FEAT_MEC);
```

Library pseudocode for shared/functions/extension/HaveFeatMOPS

```
// HaveFeatMOPS()
// ==============
// Returns TRUE if the CPY* and SET* instructions are supported, and FALSE otherwise.

boolean HaveFeatMOPS()
    return IsFeatureImplemented(FEAT_MOPS);
```

Library pseudocode for shared/functions/extension/HaveFeatNMI

```
// HaveFeatNMI()
// =============
// Returns TRUE if the Non-Maskable Interrupt extension is
// implemented, and FALSE otherwise.

boolean HaveFeatNMI()
    return IsFeatureImplemented(FEAT_NMI);
```

Library pseudocode for shared/functions/extension/HaveFeatRPRES

```
// HaveFeatRPRES()
// ===============
// Returns TRUE if reciprocal estimate implements 12-bit precision
// when FPCR.AH=1, and FALSE otherwise.

boolean HaveFeatRPRES()
    return IsFeatureImplemented(FEAT_RPRES);
```

Library pseudocode for shared/functions/extension/HaveFeatSCTLR2

```
// HaveFeatSCTLR2()
// ================
// Returns TRUE if SCTLR2 extension
// support is implemented and FALSE otherwise.

boolean HaveFeatSCTLR2()
    return IsFeatureImplemented(FEAT_SCTLR2);
```

Library pseudocode for shared/functions/extension/HaveFeatTCR2

```
// HaveFeatTCR2()
// ==============
// Returns TRUE if TCR2 extension
// support is implemented and FALSE otherwise.

boolean HaveFeatTCR2()
    return IsFeatureImplemented(FEAT_TCR2);
```

Library pseudocode for shared/functions/extension/HaveFeatTIDCP1

```
// HaveFeatTIDCP1()
// ================
// Returns TRUE if the SCTLR_EL1.TIDCP bit is implemented and the SCTLR_EL2.TIDCP bit
// is implemented if EL2 is implemented.

boolean HaveFeatTIDCP1()
    return IsFeatureImplemented(FEAT_TIDCP1);
```

Library pseudocode for shared/functions/extension/HaveFeatTRBEExt

```
// HaveFeatTRBEExt()
// =================
// Returns TRUE if the Trace Buffer Extension external mode is implemented, and FALSE otherwise.

boolean HaveFeatTRBEExt()
    return IsFeatureImplemented(FEAT_TRBE_EXT);
```

Library pseudocode for shared/functions/extension/HaveFeatWFxT

```
// HaveFeatWFxT()
// ==============
// Returns TRUE if WFET and WFIT instruction support is implemented,
// and FALSE otherwise.

boolean HaveFeatWFxT()
    return IsFeatureImplemented(FEAT_WFxT);
```

Library pseudocode for shared/functions/extension/HaveFeatXS

```
// HaveFeatXS()
// ============
// Returns TRUE if XS attribute and the TLBI and DSB instructions with nXS qualifier
// are supported, and FALSE otherwise.

boolean HaveFeatXS()
    return IsFeatureImplemented(FEAT_XS);
```

Library pseudocode for shared/functions/extension/HaveFlagFormatExt

```
// HaveFlagFormatExt()
// ===================
// Returns TRUE if flag format conversion instructions implemented.

boolean HaveFlagFormatExt()
    return IsFeatureImplemented(FEAT_FlagM2);
```

Library pseudocode for shared/functions/extension/HaveFlagManipulateExt

```
// HaveFlagManipulateExt()
// =======================
// Returns TRUE if flag manipulate instructions are implemented.

boolean HaveFlagManipulateExt()
    return IsFeatureImplemented(FEAT_FlagM);
```

Library pseudocode for shared/functions/extension/HaveFrintExt

```
// HaveFrintExt()
// ==============
// Returns TRUE if FRINT instructions are implemented.

boolean HaveFrintExt()
    return IsFeatureImplemented(FEAT_FRINTTS);
```

Library pseudocode for shared/functions/extension/HaveGCS

```
// HaveGCS()
// =========
// Returns TRUE if support for Guarded Control Stack is
// implemented, and FALSE otherwise.

boolean HaveGCS()
    return IsFeatureImplemented(FEAT_GCS);
```

Library pseudocode for shared/functions/extension/HaveGTGExt

```
// HaveGTGExt()
// ============
// Returns TRUE if support for guest translation granule size is implemented.

boolean HaveGTGExt()
    return IsFeatureImplemented(FEAT_GTG);
```

Library pseudocode for shared/functions/extension/HaveHPMDExt

```
// HaveHPMDExt()
// =============

boolean HaveHPMDExt()
    return IsFeatureImplemented(FEAT_PMUv3p1);
```

Library pseudocode for shared/functions/extension/HaveIDSExt

```
// HaveIDSExt()
// ============
// Returns TRUE if ID register handling feature is implemented.

boolean HaveIDSExt()
    return IsFeatureImplemented(FEAT_IDST);
```

Library pseudocode for shared/functions/extension/HaveIESB

```
// HaveIESB()
// ==========

boolean HaveIESB()
    return IsFeatureImplemented(FEAT_IESB);
```

Library pseudocode for shared/functions/extension/HaveInt8MatMulExt

```
// HaveInt8MatMulExt()
// ===================
// Returns TRUE if AArch64 8-bit integer matrix multiply instruction support
// implemented, and FALSE otherwise.

boolean HaveInt8MatMulExt()
    return IsFeatureImplemented(FEAT_I8MM);
```

Library pseudocode for shared/functions/extension/HaveLRCPC3Ext

```
// HaveLRCPC3Ext()
// ===============
// Returns TRUE if FEAT_LRCPC3 instructions are supported, and FALSE otherwise.

boolean HaveLRCPC3Ext()
    return IsFeatureImplemented(FEAT_LRCPC3);
```

Library pseudocode for shared/functions/extension/HaveLSE128

```
// HaveLSE128()
// ============
// Returns TRUE if LSE128 is implemented, and FALSE otherwise.

boolean HaveLSE128()
    return IsFeatureImplemented(FEAT_LSE128);
```

Library pseudocode for shared/functions/extension/HaveLSE2Ext

```
// HaveLSE2Ext()
// =============
// Returns TRUE if LSE2 is implemented, and FALSE otherwise.

boolean HaveLSE2Ext()
    return IsFeatureImplemented(FEAT_LSE2);
```

Library pseudocode for shared/functions/extension/HaveMPAMExt

```
// HaveMPAMExt()
// =============
// Returns TRUE if MPAM is implemented, and FALSE otherwise.

boolean HaveMPAMExt()
    return IsFeatureImplemented(FEAT_MPAM);
```

Library pseudocode for shared/functions/extension/HaveMPAMv0p1Ext

```
// HaveMPAMv0p1Ext()
// =================
// Returns TRUE if MPAMv0p1 is implemented, and FALSE otherwise.

boolean HaveMPAMv0p1Ext()
    return IsFeatureImplemented(FEAT_MPAMv0p1);
```

Library pseudocode for shared/functions/extension/HaveMPAMv1p1Ext

```
// HaveMPAMv1p1Ext()
// =================
// Returns TRUE if MPAMv1p1 is implemented, and FALSE otherwise.

boolean HaveMPAMv1p1Ext()
    return IsFeatureImplemented(FEAT_MPAMv1p1);
```

Library pseudocode for shared/functions/extension/HaveMTE2Ext

```
// HaveMTE2Ext()
// =============
// Returns TRUE if MTE support is beyond EL0, and FALSE otherwise.

boolean HaveMTE2Ext()
    return IsFeatureImplemented(FEAT_MTE2);
```

Library pseudocode for shared/functions/extension/HaveMTE4Ext

```
// HaveMTE4Ext()
// =============
// Returns TRUE if functionality in FEAT_MTE4 is implemented, and FALSE otherwise.

boolean HaveMTE4Ext()
    return IsFeatureImplemented(FEAT_MTE4);
```

Library pseudocode for shared/functions/extension/HaveMTEAsymFaultExt

```
// HaveMTEAsymFaultExt()
// =====================
// Returns TRUE if MTE Asymmetric Fault Handling support is
// implemented, and FALSE otherwise.

boolean HaveMTEAsymFaultExt()
    return IsFeatureImplemented(FEAT_MTE4);
```

Library pseudocode for shared/functions/extension/HaveMTEAsyncExt

```
// HaveMTEAsyncExt()
// =================
// Returns TRUE if MTE supports Asynchronous faulting, and FALSE otherwise.

boolean HaveMTEAsyncExt()
    return IsFeatureImplemented(FEAT_MTE4);
```

Library pseudocode for shared/functions/extension/HaveMTECanonicalTagCheckingExt

```
// HaveMTECanonicalTagCheckingExt()
// ================================
// Returns TRUE if MTE Canonical Tag Checking functionality is
// implemented, and FALSE otherwise.

boolean HaveMTECanonicalTagCheckingExt()
    return IsFeatureImplemented(FEAT_MTE_CANONICAL_TAGS);
```

Library pseudocode for shared/functions/extension/HaveMTEExt

```
// HaveMTEExt()
// ============
// Returns TRUE if instruction-only MTE implemented, and FALSE otherwise.

boolean HaveMTEExt()
    return IsFeatureImplemented(FEAT_MTE);
```

Library pseudocode for shared/functions/extension/HaveMTEPermExt

```
// HaveMTEPermExt()
// ================
// Returns TRUE if MTE_PERM implemented, and FALSE otherwise.

boolean HaveMTEPermExt()
    return IsFeatureImplemented(FEAT_MTE_PERM);
```

Library pseudocode for shared/functions/extension/HaveMTEStoreOnlyExt

```
// HaveMTEStoreOnlyExt()
// =====================
// Returns TRUE if MTE Store-only Tag Checking functionality is
// implemented, and FALSE otherwise.

boolean HaveMTEStoreOnlyExt()
    return IsFeatureImplemented(FEAT_MTE_STORE_ONLY);
```

Library pseudocode for shared/functions/extension/HaveNV2Ext

```
// HaveNV2Ext()
// ============
// Returns TRUE if Enhanced Nested Virtualization is implemented.

boolean HaveNV2Ext()
    return IsFeatureImplemented(FEAT_NV2);
```

Library pseudocode for shared/functions/extension/HaveNVExt

```
// HaveNVExt()
// ===========
// Returns TRUE if Nested Virtualization is implemented.

boolean HaveNVExt()
    return IsFeatureImplemented(FEAT_NV);
```

Library pseudocode for shared/functions/extension/HaveNoSecurePMUDisableOverride

```
// HaveNoSecurePMUDisableOverride()
// ================================

boolean HaveNoSecurePMUDisableOverride()
    return IsFeatureImplemented(FEAT_Debugv8p2);
```

Library pseudocode for shared/functions/extension/HaveNoninvasiveDebugAuth

```
// HaveNoninvasiveDebugAuth()
// ==========================
// Returns TRUE if the Non-invasive debug controls are implemented.

boolean HaveNoninvasiveDebugAuth()
    return !IsFeatureImplemented(FEAT_Debugv8p4);
```

Library pseudocode for shared/functions/extension/HavePAN3Ext

```
// HavePAN3Ext()
// =============
// Returns TRUE if SCTLR_EL1.EPAN and SCTLR_EL2.EPAN support is implemented,
// and FALSE otherwise.

boolean HavePAN3Ext()
    return IsFeatureImplemented(FEAT_PAN3);
```

Library pseudocode for shared/functions/extension/HavePANExt

```
// HavePANExt()
// ============

boolean HavePANExt()
    return IsFeatureImplemented(FEAT_PAN);
```

Library pseudocode for shared/functions/extension/HavePFAR

```
// HavePFAR()
// ==========
// Returns TRUE if the Physical Fault Address Extension is implemented, and FALSE
// otherwise.

boolean HavePFAR()
    return IsFeatureImplemented(FEAT_PFAR);
```

Library pseudocode for shared/functions/extension/HavePMUv3

```
// HavePMUv3()
// ===========
// Returns TRUE if the Performance Monitors extension is implemented, and FALSE otherwise.

boolean HavePMUv3()
    return IsFeatureImplemented(FEAT_PMUv3);
```

Library pseudocode for shared/functions/extension/HavePMUv3EDGE

```
// HavePMUv3EDGE()
// ===============
// Returns TRUE if support for PMU event edge detection is implemented, and FALSE otherwise.

boolean HavePMUv3EDGE()
    return IsFeatureImplemented(FEAT_PMUv3_EDGE);
```

Library pseudocode for shared/functions/extension/HavePMUv3ICNTR

```
// HavePMUv3ICNTR()
// ================
// Returns TRUE if support for the Fixed-function instruction counter is
// implemented, and FALSE otherwise.

boolean HavePMUv3ICNTR()
    return IsFeatureImplemented(FEAT_PMUv3_ICNTR);
```

Library pseudocode for shared/functions/extension/HavePMUv3TH

```
// HavePMUv3TH()
// =============
// Returns TRUE if the PMUv3 threshold extension is implemented, and FALSE otherwise.

boolean HavePMUv3TH()
    return IsFeatureImplemented(FEAT_PMUv3_TH);
```

Library pseudocode for shared/functions/extension/HavePMUv3p1

```
// HavePMUv3p1()
// =============
// Returns TRUE if the Performance Monitors extension is implemented, and FALSE otherwise.

boolean HavePMUv3p1()
    return IsFeatureImplemented(FEAT_PMUv3p1);
```

Library pseudocode for shared/functions/extension/HavePMUv3p4

```
// HavePMUv3p4()
// =============
// Returns TRUE if the PMUv3.4 extension is implemented, and FALSE otherwise.

boolean HavePMUv3p4()
    return IsFeatureImplemented(FEAT_PMUv3p4);
```

Library pseudocode for shared/functions/extension/HavePMUv3p5

```
// HavePMUv3p5()
// =============
// Returns TRUE if the PMUv3.5 extension is implemented, and FALSE otherwise.

boolean HavePMUv3p5()
    return IsFeatureImplemented(FEAT_PMUv3p5);
```

Library pseudocode for shared/functions/extension/HavePMUv3p7

```
// HavePMUv3p7()
// =============
// Returns TRUE if the PMUv3.7 extension is implemented, and FALSE otherwise.

boolean HavePMUv3p7()
    return IsFeatureImplemented(FEAT_PMUv3p7);
```

Library pseudocode for shared/functions/extension/HavePMUv3p9

```
// HavePMUv3p9()
// =============
// Returns TRUE if the PMUv3.9 extension is implemented, and FALSE otherwise.

boolean HavePMUv3p9()
    return IsFeatureImplemented(FEAT_PMUv3p9);
```

Library pseudocode for shared/functions/extension/HavePageBasedHardwareAttributes

```
// HavePageBasedHardwareAttributes()
// =================================

boolean HavePageBasedHardwareAttributes()
    return IsFeatureImplemented(FEAT_HPDS2);
```

Library pseudocode for shared/functions/extension/HaveQRDMLAHExt

```
// HaveQRDMLAHExt()
// ================

boolean HaveQRDMLAHExt()
    return IsFeatureImplemented(FEAT_RDM);
```

Library pseudocode for shared/functions/extension/HaveRASExt

```
// HaveRASExt()
// ============

boolean HaveRASExt()
    return IsFeatureImplemented(FEAT_RAS);
```

Library pseudocode for shared/functions/extension/HaveRASv2Ext

```
// HaveRASv2Ext()
// ==============
// Returns TRUE if support for RASv2 is implemented, and FALSE otherwise.

boolean HaveRASv2Ext()
    return IsFeatureImplemented(FEAT_RASv2);
```

Library pseudocode for shared/functions/extension/HaveRME

```
// HaveRME()
// =========
// Returns TRUE if the Realm Management Extension is implemented, and FALSE
// otherwise.

boolean HaveRME()
    return IsFeatureImplemented(FEAT_RME);
```

Library pseudocode for shared/functions/extension/HaveRNG

```
// HaveRNG()
// =========
// Returns TRUE if Random Number Generator extension
// support is implemented and FALSE otherwise.

boolean HaveRNG()
    return IsFeatureImplemented(FEAT_RNG);
```

Library pseudocode for shared/functions/extension/HaveS1PIExt

```
// HaveS1PIExt()
// =============
// Returns TRUE if the S1 Permission Indirection extension is
// implemented and FALSE otherwise.

boolean HaveS1PIExt()
    return IsFeatureImplemented(FEAT_S1PIE);
```

Library pseudocode for shared/functions/extension/HaveS1POExt

```
// HaveS1POExt()
// =============
// Returns TRUE if the S1 Permission Overlay extension is
// implemented and FALSE otherwise.

boolean HaveS1POExt()
    return IsFeatureImplemented(FEAT_S1POE);
```

Library pseudocode for shared/functions/extension/HaveS2PIExt

```
// HaveS2PIExt()
// =============
// Returns TRUE if the S2 Permission Indirection extension is
// implemented and FALSE otherwise.

boolean HaveS2PIExt()
    return IsFeatureImplemented(FEAT_S2PIE);
```

Library pseudocode for shared/functions/extension/HaveS2POExt

```
// HaveS2POExt()
// =============
// Returns TRUE if the S2 Permission Overlay extension is
// implemented and FALSE otherwise.

boolean HaveS2POExt()
    return IsFeatureImplemented(FEAT_S2POE);
```

Library pseudocode for shared/functions/extension/HaveSBExt

```
// HaveSBExt()
// ===========
// Returns TRUE if support for SB is implemented, and FALSE otherwise.

boolean HaveSBExt()
    return IsFeatureImplemented(FEAT_SB);
```

Library pseudocode for shared/functions/extension/HaveSSBSExt

```
// HaveSSBSExt()
// =============
// Returns TRUE if support for SSBS is implemented, and FALSE otherwise.

boolean HaveSSBSExt()
    return IsFeatureImplemented(FEAT_SSBS);
```

Library pseudocode for shared/functions/extension/HaveSecureEL2Ext

```
// HaveSecureEL2Ext()
// ==================
// Returns TRUE if Secure EL2 is implemented.

boolean HaveSecureEL2Ext()
    return IsFeatureImplemented(FEAT_SEL2);
```

Library pseudocode for shared/functions/extension/HaveSecureExtDebugView

```
// HaveSecureExtDebugView()
// ========================
// Returns TRUE if support for Secure and Non-secure views of debug peripherals
// is implemented.

boolean HaveSecureExtDebugView()
    return IsFeatureImplemented(FEAT_Debugv8p4);
```

Library pseudocode for shared/functions/extension/HaveSelfHostedTrace

```
// HaveSelfHostedTrace()
// =====================

boolean HaveSelfHostedTrace()
    return IsFeatureImplemented(FEAT_TRF);
```

Library pseudocode for shared/functions/extension/HaveSmallTranslationTblExt

```
// HaveSmallTranslationTblExt()
// ============================
// Returns TRUE if Small Translation Table Support is implemented.

boolean HaveSmallTranslationTableExt()
    return IsFeatureImplemented(FEAT_TTST);
```

Library pseudocode for shared/functions/extension/HaveSoftwareLock

```
// HaveSoftwareLock()
// ==================
// Returns TRUE if Software Lock is implemented.

boolean HaveSoftwareLock(Component component)
    if Havev8p4Debug() then
        return FALSE;
    if HaveDoPD() && component != Component_CTI then
        return FALSE;
    case component of
        when Component_Debug
            return boolean IMPLEMENTATION_DEFINED "Debug has Software Lock";
        when Component_PMU
            return boolean IMPLEMENTATION_DEFINED "PMU has Software Lock";
        when Component_CTI
            return boolean IMPLEMENTATION_DEFINED "CTI has Software Lock";
        otherwise
            Unreachable();
```

Library pseudocode for shared/functions/extension/HaveStage2MemAttrControl

```
// HaveStage2MemAttrControl()
// ==========================
// Returns TRUE if support for Stage2 control of memory types and cacheability
// attributes is implemented.

boolean HaveStage2MemAttrControl()
    return IsFeatureImplemented(FEAT_S2FWB);
```

Library pseudocode for shared/functions/extension/HaveStatisticalProfiling

```
// HaveStatisticalProfiling()
// ==========================
// Returns TRUE if Statistical Profiling Extension is implemented,
// and FALSE otherwise.

boolean HaveStatisticalProfiling()
    return IsFeatureImplemented(FEAT_SPE);
```

Library pseudocode for shared/functions/extension/HaveStatisticalProfilingFDS

```
// HaveStatisticalProfilingFDS()
// =============================
// Returns TRUE if the SPE_FDS extension is implemented, and FALSE otherwise.

boolean HaveStatisticalProfilingFDS()
    return IsFeatureImplemented(FEAT_SPE_FDS);
```

Library pseudocode for shared/functions/extension/HaveStatisticalProfilingv1p1

```
// HaveStatisticalProfilingv1p1()
// ==============================
// Returns TRUE if the SPEv1p1 extension is implemented, and FALSE otherwise.

boolean HaveStatisticalProfilingv1p1()
    return IsFeatureImplemented(FEAT_SPEv1p1);
```

Library pseudocode for shared/functions/extension/HaveStatisticalProfilingv1p2

```
// HaveStatisticalProfilingv1p2()
// ==============================
// Returns TRUE if the SPEv1p2 extension is implemented, and FALSE otherwise.

boolean HaveStatisticalProfilingv1p2()
    return IsFeatureImplemented(FEAT_SPEv1p2);
```

Library pseudocode for shared/functions/extension/HaveStatisticalProfilingv1p4

```
// HaveStatisticalProfilingv1p4()
// ==============================
// Returns TRUE if the SPEv1p4 extension is implemented, and FALSE otherwise.

boolean HaveStatisticalProfilingv1p4()
    return IsFeatureImplemented(FEAT_SPEv1p4);
```

Library pseudocode for shared/functions/extension/HaveSysInstr128

```
// HaveSysInstr128()
// =================
// Returns TRUE if support for System Instructions that can
// take 128-bit inputs is implemented, and FALSE otherwise.

boolean HaveSysInstr128()
    return IsFeatureImplemented(FEAT_SYSINSTR128);
```

Library pseudocode for shared/functions/extension/HaveSysReg128

```
// HaveSysReg128()
// ===============
// Returns TRUE if support for 128-bit System Registers is implemented, and FALSE otherwise.

boolean HaveSysReg128()
    return IsFeatureImplemented(FEAT_SYSREG128);
```

Library pseudocode for shared/functions/extension/HaveTHExt

```
// HaveTHExt()
// ===========
// Returns TRUE if support for Translation Hardening Extension is implemented.

boolean HaveTHExt()
    return IsFeatureImplemented(FEAT_THE);
```

Library pseudocode for shared/functions/extension/HaveTME

```
// HaveTME()
// =========

boolean HaveTME()
    return IsFeatureImplemented(FEAT_TME);
```

Library pseudocode for shared/functions/extension/HaveTWEDExt

```
// HaveTWEDExt()
// =============
// Returns TRUE if Delayed Trapping of WFE instruction support is implemented,
// and FALSE otherwise.

boolean HaveTWEDExt()
    return IsFeatureImplemented(FEAT_TWED);
```

Library pseudocode for shared/functions/extension/HaveTraceBufferExtension

```
// HaveTraceBufferExtension()
// ==========================
// Returns TRUE if Trace Buffer Extension is implemented, and FALSE otherwise.

boolean HaveTraceBufferExtension()
    return IsFeatureImplemented(FEAT_TRBE);
```

Library pseudocode for shared/functions/extension/HaveTraceExt

```
// HaveTraceExt()
// ==============
// Returns TRUE if Trace functionality as described by the Trace Architecture
// is implemented.

boolean HaveTraceExt()
    return boolean IMPLEMENTATION_DEFINED "Has Trace Architecture functionality";
```

Library pseudocode for shared/functions/extension/HaveTrapLoadStoreMultipleDeviceExt

```
// HaveTrapLoadStoreMultipleDeviceExt()
// ====================================

boolean HaveTrapLoadStoreMultipleDeviceExt()
    return IsFeatureImplemented(FEAT_LSMAOC);
```

Library pseudocode for shared/functions/extension/HaveUAOExt

```
// HaveUAOExt()
// ============

boolean HaveUAOExt()
    return IsFeatureImplemented(FEAT_UAO);
```

Library pseudocode for shared/functions/extension/HaveV82Debug

```
// HaveV82Debug()
// ==============

boolean HaveV82Debug()
    return IsFeatureImplemented(FEAT_Debugv8p2);
```

Library pseudocode for shared/functions/extension/HaveVirtHostExt

```
// HaveVirtHostExt()
// =================

boolean HaveVirtHostExt()
    return IsFeatureImplemented(FEAT_VHE);
```

Library pseudocode for shared/functions/extension/Havev8p4Debug

```
// Havev8p4Debug()
// ===============
// Returns TRUE if support for the Debugv8p4 feature is implemented and FALSE otherwise.

boolean Havev8p4Debug()
    return IsFeatureImplemented(FEAT_Debugv8p4);
```

Library pseudocode for shared/functions/extension/Havev8p8Debug

```
// Havev8p8Debug()
// ===============
// Returns TRUE if support for the Debugv8p8 feature is implemented and FALSE otherwise.

boolean Havev8p8Debug()
    return IsFeatureImplemented(FEAT_Debugv8p8);
```

Library pseudocode for shared/functions/extension/Havev8p9Debug

```
// Havev8p9Debug()
// ===============
// Returns TRUE if support for the Debugv8p9 feature is implemented, and FALSE otherwise.

boolean Havev8p9Debug()
    return IsFeatureImplemented(FEAT_Debugv8p9);
```

Library pseudocode for shared/functions/extension/InsertIESBBeforeException

```
// InsertIESBBeforeException()
// ===========================
// Returns an implementation defined choice whether to insert an implicit error synchronization
// barrier before exception.
// If SCTLR_ELx.IESB is 1 when an exception is generated to ELx, any pending Unrecoverable
// SError interrupt must be taken before executing any instructions in the exception handler.
// However, this can be before the branch to the exception handler is made.

boolean InsertIESBBeforeException(bits(2) el)
    return (HaveIESB() && boolean IMPLEMENTATION_DEFINED
            "Has Implicit Error Synchronization Barrier before Exception");
```

Library pseudocode for shared/functions/extension/IsG1ActivityMonitorImplemented

```
// IsG1ActivityMonitorImplemented()
// ================================
// Returns TRUE if a G1 activity monitor is implemented for the counter
// and FALSE otherwise.

boolean IsG1ActivityMonitorImplemented(integer i);
```

Library pseudocode for shared/functions/extension/IsG1ActivityMonitorOffsetImplemented

```
// IsG1ActivityMonitorOffsetImplemented()
// ======================================
// Returns TRUE if a G1 activity monitor offset is implemented for the counter,
// and FALSE otherwise.

boolean IsG1ActivityMonitorOffsetImplemented(integer i);
```

Library pseudocode for shared/functions/externalaborts/AArch32.PEErrorState

```
// AArch32.PEErrorState()
// ======================
// Returns the error state by PE on taking an SError Interrupt
// to AArch32 level.

ErrorState AArch32.PEErrorState(FaultRecord fault)
    if (!ErrorIsContained() ||
        (!ErrorIsSynchronized() && !StateIsRecoverable()) ||
         ReportErrorAsUC()) then
        return ErrorState_UC;

    if !StateIsRecoverable() || ReportErrorAsUEU() then
        return ErrorState_UEU;

    if ActionRequired() || ReportErrorAsUER() then
        return ErrorState_UER;

    return ErrorState_UEO;
```

Library pseudocode for shared/functions/externalaborts/AArch64.PEErrorState

```
// AArch64.PEErrorState()
// ======================
// Returns the error state by PE on taking a Synchronous
// or Asynchronous exception.

ErrorState AArch64.PEErrorState(FaultRecord fault)
    if !IsExternalSyncAbort(fault) && ExtAbortToA64(fault) then
        if ReportErrorAsUncategorized() then
            return ErrorState_Uncategorized;
        if ReportErrorAsIMPDEF() then
            return ErrorState_IMPDEF;

    assert !FaultIsCorrected();
    if (!ErrorIsContained() ||
        (!ErrorIsSynchronized() && !StateIsRecoverable()) ||
         ReportErrorAsUC()) then
        return ErrorState_UC;

    if !StateIsRecoverable() || ReportErrorAsUEU() then
        if IsExternalSyncAbort(fault) then  // Implies taken to AArch64
            return ErrorState_UC;
        else
            return ErrorState_UEU;

    if (ActionRequired() || ReportErrorAsUER()) then
        return ErrorState_UER;

    return ErrorState_UEO;
```

Library pseudocode for shared/functions/externalaborts/ActionRequired

```
// ActionRequired()
// ================
// Return an implementation specific value:
// returns TRUE if action is required, FALSE otherwise.

boolean ActionRequired();
```

Library pseudocode for shared/functions/externalaborts/ClearPendingPhysicalSError

```
// ClearPendingPhysicalSError()
// ============================
// Clear a pending physical SError interrupt.

ClearPendingPhysicalSError();
```

Library pseudocode for shared/functions/externalaborts/ClearPendingVirtualSError

```
// ClearPendingVirtualSError()
// ===========================
// Clear a pending virtual SError interrupt.

ClearPendingVirtualSError()
    if ELUsingAArch32(EL2) then
        HCR.VA = '0';
    else
        HCR_EL2.VSE = '0';
```

Library pseudocode for shared/functions/externalaborts/ErrorIsContained

```
// ErrorIsContained()
// ==================
// Return an implementation specific value:
// TRUE if Error is contained by the PE, FALSE otherwise.

boolean ErrorIsContained();
```

Library pseudocode for shared/functions/externalaborts/ErrorIsSynchronized

```
// ErrorIsSynchronized()
// =====================
// Return an implementation specific value:
// returns TRUE if Error is synchronized by any synchronization event
// FALSE otherwise.

boolean ErrorIsSynchronized();
```

Library pseudocode for shared/functions/externalaborts/ExtAbortToA64

```
// ExtAbortToA64()
// ===============
// Returns TRUE if synchronous exception is being taken to A64 exception
// level.

boolean ExtAbortToA64(FaultRecord fault)
    // Check if routed to AArch64 state
    route_to_aarch64 = PSTATE.EL == EL0 && !ELUsingAArch32(EL1);

    if !route_to_aarch64 && EL2Enabled() && !ELUsingAArch32(EL2) then
        route_to_aarch64 = (HCR_EL2.TGE == '1' || IsSecondStage(fault) ||
                            (HaveRASExt() && HCR_EL2.TEA == '1' && IsExternalAbort(fault)) ||
                            (IsDebugException(fault) && MDCR_EL2.TDE == '1'));

    if !route_to_aarch64 && HaveEL(EL3) && !ELUsingAArch32(EL3) then
        route_to_aarch64 = SCR_GEN[].EA == '1' && IsExternalAbort(fault);

    return route_to_aarch64 && IsExternalSyncAbort(fault.statuscode);
```

Library pseudocode for shared/functions/externalaborts/FaultIsCorrected

```
// FaultIsCorrected()
// ==================
// Return an implementation specific value:
// TRUE if fault is corrected by the PE, FALSE otherwise.

boolean FaultIsCorrected();
```

Library pseudocode for shared/functions/externalaborts/GetPendingPhysicalSError

```
// GetPendingPhysicalSError()
// ==========================
// Returns the FaultRecord containing details of pending Physical SError
// interrupt.

FaultRecord GetPendingPhysicalSError();
```

Library pseudocode for shared/functions/externalaborts/HandleExternalAbort

```
// HandleExternalAbort()
// =====================
// Takes a Synchronous/Asynchronous abort based on fault.

HandleExternalAbort(PhysMemRetStatus memretstatus, boolean iswrite,
                    AddressDescriptor memaddrdesc, integer size,
                    AccessDescriptor accdesc)
    assert (memretstatus.statuscode IN {Fault_SyncExternal, Fault_AsyncExternal} ||
           (!HaveRASExt() && memretstatus.statuscode IN {Fault_SyncParity,
                                                          Fault_AsyncParity}));

    fault = NoFault(accdesc);
    fault.statuscode = memretstatus.statuscode;
    fault.write      = iswrite;
    fault.extflag    = memretstatus.extflag;
    // It is implementation specific whether External aborts signaled
    // in-band synchronously are taken synchronously or asynchronously
    if (IsExternalSyncAbort(fault) &&
            !IsExternalAbortTakenSynchronously(memretstatus, iswrite, memaddrdesc,
                                               size, accdesc)) then
        if fault.statuscode == Fault_SyncParity then
            fault.statuscode = Fault_AsyncParity;
        else
            fault.statuscode = Fault_AsyncExternal;

    if HaveRASExt() then
        fault.merrorstate = memretstatus.merrorstate;

    if IsExternalSyncAbort(fault) then
        if UsingAArch32() then
            AArch32.Abort(memaddrdesc.vaddress<31:0>, fault);
        else
            AArch64.Abort(memaddrdesc.vaddress, fault);

    else
        PendSErrorInterrupt(fault);
```

Library pseudocode for shared/functions/externalaborts/HandleExternalReadAbort

```
// HandleExternalReadAbort()
// =========================
// Wrapper function for HandleExternalAbort function in case of an External
// Abort on memory read.

HandleExternalReadAbort(PhysMemRetStatus memstatus, AddressDescriptor memaddrdesc,
                        integer size, AccessDescriptor accdesc)
    iswrite = FALSE;
    HandleExternalAbort(memstatus, iswrite, memaddrdesc, size, accdesc);
```

Library pseudocode for shared/functions/externalaborts/HandleExternalTTWAbort

```
// HandleExternalTTWAbort()
// ========================
// Take Asynchronous abort or update FaultRecord for Translation Table Walk
// based on PhysMemRetStatus.

FaultRecord HandleExternalTTWAbort(PhysMemRetStatus memretstatus, boolean iswrite,
                                   AddressDescriptor memaddrdesc,
                                   AccessDescriptor accdesc, integer size,
                                   FaultRecord input_fault)
    output_fault = input_fault;
    output_fault.extflag = memretstatus.extflag;
    output_fault.statuscode = memretstatus.statuscode;
    if (IsExternalSyncAbort(output_fault) &&
            !IsExternalAbortTakenSynchronously(memretstatus, iswrite, memaddrdesc,
                                               size, accdesc)) then
        if output_fault.statuscode == Fault_SyncParity then
            output_fault.statuscode = Fault_AsyncParity;
        else
            output_fault.statuscode = Fault_AsyncExternal;

    // If a synchronous fault is on a translation table walk, then update
    // the fault type
    if IsExternalSyncAbort(output_fault) then
        if output_fault.statuscode == Fault_SyncParity then
            output_fault.statuscode = Fault_SyncParityOnWalk;
        else
            output_fault.statuscode = Fault_SyncExternalOnWalk;
    if HaveRASExt() then
        output_fault.merrorstate = memretstatus.merrorstate;
    if !IsExternalSyncAbort(output_fault) then
        PendSErrorInterrupt(output_fault);
        output_fault.statuscode = Fault_None;
    return output_fault;
```

Library pseudocode for shared/functions/externalaborts/HandleExternalWriteAbort

```
// HandleExternalWriteAbort()
// ==========================
// Wrapper function for HandleExternalAbort function in case of an External
// Abort on memory write.

HandleExternalWriteAbort(PhysMemRetStatus memstatus, AddressDescriptor memaddrdesc,
                         integer size, AccessDescriptor accdesc)
    iswrite = TRUE;
    HandleExternalAbort(memstatus, iswrite, memaddrdesc, size, accdesc);
```

Library pseudocode for shared/functions/externalaborts/IsExternalAbortTakenSynchronously

```
// IsExternalAbortTakenSynchronously()
// ===================================
// Return an implementation specific value:
// TRUE if the fault returned for the access can be taken synchronously,
// FALSE otherwise.
//
// This might vary between accesses, for example depending on the error type
// or memory type being accessed.
// External aborts on data accesses and translation table walks on data accesses
// can be either synchronous or asynchronous.
//
// When FEAT_DoubleFault is not implemented, External aborts on instruction
// fetches and translation table walks on instruction fetches can be either
// synchronous or asynchronous.
// When FEAT_DoubleFault is implemented, all External abort exceptions on
// instruction fetches and translation table walks on instruction fetches
// must be synchronous.

boolean IsExternalAbortTakenSynchronously(PhysMemRetStatus memstatus,
                                          boolean iswrite,
                                          AddressDescriptor desc,
                                          integer size,
                                          AccessDescriptor accdesc);
```

Library pseudocode for shared/functions/externalaborts/IsPhysicalSErrorPending

```
// IsPhysicalSErrorPending()
// =========================
// Returns TRUE if a physical SError interrupt is pending.

boolean IsPhysicalSErrorPending();
```

Library pseudocode for shared/functions/externalaborts/IsSErrorEdgeTriggered

```
// IsSErrorEdgeTriggered()
// =======================
// Returns TRUE if the physical SError interrupt is edge-triggered
// and FALSE otherwise.

boolean IsSErrorEdgeTriggered()
    if HaveDoubleFaultExt() then
        return TRUE;
    else
        return boolean IMPLEMENTATION_DEFINED "Edge-triggered SError";
```

Library pseudocode for shared/functions/externalaborts/IsSynchronizablePhysicalSErrorPending

```
// IsSynchronizablePhysicalSErrorPending()
// =======================================
// Returns TRUE if a synchronizable physical SError interrupt is pending.

boolean IsSynchronizablePhysicalSErrorPending();
```

Library pseudocode for shared/functions/externalaborts/IsVirtualSErrorPending

```
// IsVirtualSErrorPending()
// ========================
// Return TRUE if a virtual SError interrupt is pending.

boolean IsVirtualSErrorPending()
    if ELUsingAArch32(EL2) then
        return HCR.VA == '1';
    else
        return HCR_EL2.VSE == '1';
```

Library pseudocode for shared/functions/externalaborts/PendSErrorInterrupt

```
// PendSErrorInterrupt()
// =====================
// Pend the SError Interrupt.

PendSErrorInterrupt(FaultRecord fault);
```

Library pseudocode for shared/functions/externalaborts/ReportErrorAsIMPDEF

```
// ReportErrorAsIMPDEF()
// =====================
// Return an implementation specific value:
// returns TRUE if Error is IMPDEF, FALSE otherwise.

boolean ReportErrorAsIMPDEF();
```

Library pseudocode for shared/functions/externalaborts/ReportErrorAsUC

```
// ReportErrorAsUC()
// =================
// Return an implementation specific value:
// returns TRUE if Error is Uncontainable, FALSE otherwise.

boolean ReportErrorAsUC();
```

Library pseudocode for shared/functions/externalaborts/ReportErrorAsUER

```
// ReportErrorAsUER()
// ==================
// Return an implementation specific value:
// returns TRUE if Error is Recoverable, FALSE otherwise.

boolean ReportErrorAsUER();
```

Library pseudocode for shared/functions/externalaborts/ReportErrorAsUEU

```
// ReportErrorAsUEU()
// ==================
// Return an implementation specific value:
// returns TRUE if Error is Unrecoverable, FALSE otherwise.

boolean ReportErrorAsUEU();
```

Library pseudocode for shared/functions/externalaborts/ReportErrorAsUncategorized

```
// ReportErrorAsUncategorized()
// ============================
// Return an implementation specific value:
// returns TRUE if Error is uncategorized, FALSE otherwise.

boolean ReportErrorAsUncategorized();
```

Library pseudocode for shared/functions/externalaborts/StateIsRecoverable

```
// StateIsRecoverable()
// =====================
// Return an implementation specific value:
// returns TRUE is PE State is unrecoverable else FALSE.

boolean StateIsRecoverable();
```

Library pseudocode for shared/functions/float/bfloat/BFAdd

```
// BFAdd()
// =======
// Non-widening BFloat16 addition used by SVE2 instructions.

bits(N) BFAdd(bits(N) op1, bits(N) op2, FPCRType fpcr)
    boolean fpexc = TRUE;
    return BFAdd(op1, op2, fpcr, fpexc);

// BFAdd()
// =======
// Non-widening BFloat16 addition following computational behaviors
// corresponding to instructions that read and write BFloat16 values.
// Calculates op1 + op2.
// The 'fpcr' argument supplies the FPCR control bits.

bits(N) BFAdd(bits(N) op1, bits(N) op2, FPCRType fpcr, boolean fpexc)

    assert N == 16;
    FPRounding rounding = FPRoundingMode(fpcr);
    boolean done;
    bits(2*N) result;

    bits(2*N) op1_s = op1 : Zeros(N);
    bits(2*N) op2_s = op2 : Zeros(N);
    (type1,sign1,value1) = FPUnpack(op1_s, fpcr, fpexc);
    (type2,sign2,value2) = FPUnpack(op2_s, fpcr, fpexc);

    (done,result) = FPProcessNaNs(type1, type2, op1_s, op2_s, fpcr, fpexc);

    if !done then
        inf1 = (type1 == FPType_Infinity);
        inf2 = (type2 == FPType_Infinity);
        zero1 = (type1 == FPType_Zero);
        zero2 = (type2 == FPType_Zero);

        if inf1 && inf2 && sign1 == NOT(sign2) then
            result = FPDefaultNaN(fpcr, 2*N);
            if fpexc then FPProcessException(FPExc_InvalidOp, fpcr);
        elsif (inf1 && sign1 == '0') || (inf2 && sign2 == '0') then
            result = FPInfinity('0', 2*N);
        elsif (inf1 && sign1 == '1') || (inf2 && sign2 == '1') then
            result = FPInfinity('1', 2*N);
        elsif zero1 && zero2 && sign1 == sign2 then
            result = FPZero(sign1, 2*N);
        else
            result_value = value1 + value2;
            if result_value == 0.0 then  // Sign of exact zero result depends on rounding mode
                result_sign = if rounding == FPRounding_NEGINF then '1' else '0';
                result = FPZero(result_sign, 2*N);
            else
                result = FPRoundBF(result_value, fpcr, rounding, fpexc, 2*N);

        if fpexc then FPProcessDenorms(type1, type2, 2*N, fpcr);

    return result<2*N-1:N>;
```

Library pseudocode for shared/functions/float/bfloat/BFAdd_ZA

```
// BFAdd_ZA()
// ==========
// Non-widening BFloat16 addition used by SME2 ZA-targeting instructions.

bits(N) BFAdd_ZA(bits(N) op1, bits(N) op2, FPCRType fpcr_in)
    boolean fpexc = FALSE;
    FPCRType fpcr = fpcr_in;
    fpcr.DN = '1';          // Generate default NaN values
    return BFAdd(op1, op2, fpcr, fpexc);
```

Library pseudocode for shared/functions/float/bfloat/BFDotAdd

```
// BFDotAdd()
// ==========
// BFloat16 2-way dot-product and add to single-precision
// result = addend + op1_a*op2_a + op1_b*op2_b

bits(N) BFDotAdd(bits(N) addend, bits(N DIV 2) op1_a, bits(N DIV 2) op1_b,
                  bits(N DIV 2) op2_a, bits(N DIV 2) op2_b, FPCRType fpcr_in)
    assert N == 32;
    FPCRType fpcr = fpcr_in;

    bits(N) prod;

    bits(N) result;
    if !HaveEBF16() || fpcr.EBF == '0' then   // Standard BFloat16 behaviors
        prod = FPAdd_BF16(BFMulH(op1_a, op2_a), BFMulH(op1_b, op2_b));
        result = FPAdd_BF16(addend, prod);
    else                                       // Extended BFloat16 behaviors
        boolean isbfloat16 = TRUE;
        boolean fpexc = FALSE;  // Do not generate floating-point exceptions
        fpcr.DN = '1';          // Generate default NaN values
        prod = FPDot(op1_a, op1_b, op2_a, op2_b, fpcr, isbfloat16, fpexc);
        result = FPAdd(addend, prod, fpcr, fpexc);

    return result;
```

Library pseudocode for shared/functions/float/bfloat/BFInfinity

```
// BFInfinity()
// ============

bits(N) BFInfinity(bit sign, integer N)
    assert N == 16;
    constant integer E = 8;
    constant integer F = N - (E + 1);
    return sign : Ones(E) : Zeros(F);
```

Library pseudocode for shared/functions/float/bfloat/BFMatMulAdd

```
// BFMatMulAdd()
// =============
// BFloat16 matrix multiply and add to single-precision matrix
// result[2, 2] = addend[2, 2] + (op1[2, 4] * op2[4, 2])

bits(N) BFMatMulAdd(bits(N) addend, bits(N) op1, bits(N) op2)

    assert N == 128;

    bits(N) result;
    bits(32) sum;

    for i = 0 to 1
        for j = 0 to 1
            sum = Elem[addend, 2*i + j, 32];
            for k = 0 to 1
                bits(16) elt1_a = Elem[op1, 4*i + 2*k + 0, 16];
                bits(16) elt1_b = Elem[op1, 4*i + 2*k + 1, 16];
                bits(16) elt2_a = Elem[op2, 4*j + 2*k + 0, 16];
                bits(16) elt2_b = Elem[op2, 4*j + 2*k + 1, 16];
                sum = BFDotAdd(sum, elt1_a, elt1_b, elt2_a, elt2_b, FPCR[]);
            Elem[result, 2*i + j, 32] = sum;

    return result;
```

Library pseudocode for shared/functions/float/bfloat/BFMax

```
// BFMax()
// =======
// BFloat16 maximum.

bits(N) BFMax(bits(N) op1, bits(N) op2, FPCRType fpcr)
    boolean altfp = HaveAltFP() && !UsingAArch32() && fpcr.AH == '1';
    return BFMax(op1, op2, fpcr, altfp);

// BFMax()
// =======
// BFloat16 maximum following computational behaviors
// corresponding to instructions that read and write BFloat16 values.
// Compare op1 and op2 and return the larger value after rounding.
// The 'fpcr' argument supplies the FPCR control bits and 'altfp' determines
// if the function should use alternative floating-point behavior.

bits(N) BFMax(bits(N) op1, bits(N) op2, FPCRType fpcr_in, boolean altfp)

    assert N == 16;
    FPCRType fpcr = fpcr_in;
    boolean fpexc = TRUE;
    FPRounding rounding = FPRoundingMode(fpcr);
    boolean done;
    bits(2*N) result;

    bits(2*N) op1_s = op1 : Zeros(N);
    bits(2*N) op2_s = op2 : Zeros(N);
    (type1,sign1,value1) = FPUnpack(op1_s, fpcr, fpexc);
    (type2,sign2,value2) = FPUnpack(op2_s, fpcr, fpexc);

    if altfp && type1 == FPType_Zero && type2 == FPType_Zero && sign1 != sign2 then
        // Alternate handling of zeros with differing sign
        return BFZero(sign2, N);
    elsif altfp && (type1 IN {FPType_SNaN, FPType_QNaN} || type2 IN {FPType_SNaN, FPType_QNaN}) then
        // Alternate handling of NaN inputs
        FPProcessException(FPExc_InvalidOp, fpcr);
        return (if type2 == FPType_Zero then BFZero(sign2, N) else op2);

    (done,result) = FPProcessNaNs(type1, type2, op1_s, op2_s, fpcr);
    if !done then
        FPType fptype;
        bit sign;
        real value;
        if value1 > value2 then
            (fptype,sign,value) = (type1,sign1,value1);
        else
            (fptype,sign,value) = (type2,sign2,value2);
        if fptype == FPType_Infinity then
            result = FPInfinity(sign, 2*N);
        elsif fptype == FPType_Zero then
            sign = sign1 AND sign2;              // Use most positive sign
            result = FPZero(sign, 2*N);
        else
            if altfp then    // Denormal output is not flushed to zero
                fpcr.FZ = '0';
            result = FPRoundBF(value, fpcr, rounding, fpexc, 2*N);

        if fpexc then FPProcessDenorms(type1, type2, 2*N, fpcr);

    return result<2*N-1:N>;
```

Library pseudocode for shared/functions/float/bfloat/BFMaxNum

```
// BFMaxNum()
// ==========
// BFloat16 maximum number following computational behaviors corresponding
// to instructions that read and write BFloat16 values.
// Compare op1 and op2 and return the smaller number operand after rounding.
// The 'fpcr' argument supplies the FPCR control bits.

bits(N) BFMaxNum(bits(N) op1_in, bits(N) op2_in, FPCRType fpcr)

    assert N == 16;
    boolean fpexc = TRUE;
    boolean isbfloat16 = TRUE;
    bits(N) op1 = op1_in;
    bits(N) op2 = op2_in;
    boolean altfp = HaveAltFP() && !UsingAArch32() && fpcr.AH == '1';
    bits(N) result;

    (type1,-,-) = FPUnpackBase(op1, fpcr, fpexc, isbfloat16);
    (type2,-,-) = FPUnpackBase(op2, fpcr, fpexc, isbfloat16);

    boolean type1_nan = type1 IN {FPType_QNaN, FPType_SNaN};
    boolean type2_nan = type2 IN {FPType_QNaN, FPType_SNaN};

    if !(altfp && type1_nan && type2_nan) then
        // Treat a single quiet-NaN as -Infinity.
        if type1 == FPType_QNaN && type2 != FPType_QNaN then
            op1 = BFInfinity('1', N);
        elsif type1 != FPType_QNaN && type2 == FPType_QNaN then
            op2 = BFInfinity('1', N);

    boolean altfmaxfmin = FALSE;    // Do not use alternate NaN handling
    result = BFMax(op1, op2, fpcr, altfmaxfmin);

    return result;
```

Library pseudocode for shared/functions/float/bfloat/BFMin

```
// BFMin()
// =======
// BFloat16 minimum.

bits(N) BFMin(bits(N) op1, bits(N) op2, FPCRType fpcr)
    boolean altfp = HaveAltFP() && !UsingAArch32() && fpcr.AH == '1';
    return BFMin(op1, op2, fpcr, altfp);

// BFMin()
// =======
// BFloat16 minimum following computational behaviors
// corresponding to instructions that read and write BFloat16 values.
// Compare op1 and op2 and return the smaller value after rounding.
// The 'fpcr' argument supplies the FPCR control bits and 'altfp' determines
// if the function should use alternative floating-point behavior.

bits(N) BFMin(bits(N) op1, bits(N) op2, FPCRType fpcr_in, boolean altfp)

    assert N == 16;
    FPCRType fpcr = fpcr_in;
    boolean fpexc = TRUE;
    FPRounding rounding = FPRoundingMode(fpcr);
    boolean done;
    bits(2*N) result;

    bits(2*N) op1_s = op1 : Zeros(N);
    bits(2*N) op2_s = op2 : Zeros(N);
    (type1,sign1,value1) = FPUnpack(op1_s, fpcr, fpexc);
    (type2,sign2,value2) = FPUnpack(op2_s, fpcr, fpexc);

    if altfp && type1 == FPType_Zero && type2 == FPType_Zero && sign1 != sign2 then
        // Alternate handling of zeros with differing sign
        return BFZero(sign2, N);
    elsif altfp && (type1 IN {FPType_SNaN, FPType_QNaN} || type2 IN {FPType_SNaN, FPType_QNaN}) then
        // Alternate handling of NaN inputs
        FPProcessException(FPExc_InvalidOp, fpcr);
        return (if type2 == FPType_Zero then BFZero(sign2, N) else op2);

    (done,result) = FPProcessNaNs(type1, type2, op1_s, op2_s, fpcr);
    if !done then
        FPType fptype;
        bit sign;
        real value;
        if value1 < value2 then
            (fptype,sign,value) = (type1,sign1,value1);
        else
            (fptype,sign,value) = (type2,sign2,value2);
        if fptype == FPType_Infinity then
            result = FPInfinity(sign, 2*N);
        elsif fptype == FPType_Zero then
            sign = sign1 OR sign2;              // Use most negative sign
            result = FPZero(sign, 2*N);
        else
            if altfp then    // Denormal output is not flushed to zero
                fpcr.FZ = '0';
            result = FPRoundBF(value, fpcr, rounding, fpexc, 2*N);

        if fpexc then FPProcessDenorms(type1, type2, 2*N, fpcr);

    return result<2*N-1:N>;
```

Library pseudocode for shared/functions/float/bfloat/BFMinNum

```
// BFMinNum()
// ==========
// BFloat16 minimum number following computational behaviors corresponding
// to instructions that read and write BFloat16 values.
// Compare op1 and op2 and return the smaller number operand after rounding.
// The 'fpcr' argument supplies the FPCR control bits.

bits(N) BFMinNum(bits(N) op1_in, bits(N) op2_in, FPCRType fpcr)

    assert N == 16;
    boolean fpexc = TRUE;
    boolean isbfloat16 = TRUE;
    bits(N) op1 = op1_in;
    bits(N) op2 = op2_in;
    boolean altfp = HaveAltFP() && !UsingAArch32() && fpcr.AH == '1';
    bits(N) result;

    (type1,-,-) = FPUnpackBase(op1, fpcr, fpexc, isbfloat16);
    (type2,-,-) = FPUnpackBase(op2, fpcr, fpexc, isbfloat16);

    boolean type1_nan = type1 IN {FPType_QNaN, FPType_SNaN};
    boolean type2_nan = type2 IN {FPType_QNaN, FPType_SNaN};

    if !(altfp && type1_nan && type2_nan) then
        // Treat a single quiet-NaN as +Infinity.
        if type1 == FPType_QNaN && type2 != FPType_QNaN then
            op1 = BFInfinity('0', N);
        elsif type1 != FPType_QNaN && type2 == FPType_QNaN then
            op2 = BFInfinity('0', N);

    boolean altfmaxfmin = FALSE;    // Do not use alternate NaN handling
    result = BFMin(op1, op2, fpcr, altfmaxfmin);

    return result;
```

Library pseudocode for shared/functions/float/bfloat/BFMul

```
// BFMul()
// =======
// Non-widening BFloat16 multiply used by SVE2 instructions.

bits(N) BFMul(bits(N) op1, bits(N) op2, FPCRType fpcr)
    boolean fpexc = TRUE;
    return BFMul(op1, op2, fpcr, fpexc);

// BFMul()
// =======
// Non-widening BFloat16 multiply following computational behaviors
// corresponding to instructions that read and write BFloat16 values.
// Calculates op1 * op2.
// The 'fpcr' argument supplies the FPCR control bits.

bits(N) BFMul(bits(N) op1, bits(N) op2, FPCRType fpcr, boolean fpexc)

    assert N == 16;
    FPRounding rounding = FPRoundingMode(fpcr);
    boolean done;
    bits(2*N) result;

    bits(2*N) op1_s = op1 : Zeros(N);
    bits(2*N) op2_s = op2 : Zeros(N);
    (type1,sign1,value1) = FPUnpack(op1_s, fpcr, fpexc);
    (type2,sign2,value2) = FPUnpack(op2_s, fpcr, fpexc);

    (done,result) = FPProcessNaNs(type1, type2, op1_s, op2_s, fpcr, fpexc);

    if !done then
        inf1 = (type1 == FPType_Infinity);
        inf2 = (type2 == FPType_Infinity);
        zero1 = (type1 == FPType_Zero);
        zero2 = (type2 == FPType_Zero);

        if (inf1 && zero2) || (zero1 && inf2) then
            result = FPDefaultNaN(fpcr, 2*N);
            if fpexc then FPProcessException(FPExc_InvalidOp, fpcr);
        elsif inf1 || inf2 then
            result = FPInfinity(sign1 EOR sign2, 2*N);
        elsif zero1 || zero2 then
            result = FPZero(sign1 EOR sign2, 2*N);
        else
            result = FPRoundBF(value1*value2, fpcr, rounding, fpexc, 2*N);

        if fpexc then FPProcessDenorms(type1, type2, 2*N, fpcr);

    return result<2*N-1:N>;
```

Library pseudocode for shared/functions/float/bfloat/BFMulAdd

```
// BFMulAdd()
// ==========
// Non-widening BFloat16 fused multiply-add used by SVE2 instructions.

bits(N) BFMulAdd(bits(N) addend, bits(N) op1, bits(N) op2, FPCRType fpcr)
    boolean fpexc = TRUE;
    return BFMulAdd(addend, op1, op2, fpcr, fpexc);

// BFMulAdd()
// ==========
// Non-widening BFloat16 fused multiply-add following computational behaviors
// corresponding to instructions that read and write BFloat16 values.
// Calculates addend + op1*op2 with a single rounding.
// The 'fpcr' argument supplies the FPCR control bits.

bits(N) BFMulAdd(bits(N) addend, bits(N) op1, bits(N) op2, FPCRType fpcr, boolean fpexc)

    assert N == 16;
    FPRounding rounding = FPRoundingMode(fpcr);
    boolean done;
    bits(2*N) result;

    bits(2*N) addend_s = addend : Zeros(N);
    bits(2*N) op1_s = op1 : Zeros(N);
    bits(2*N) op2_s = op2 : Zeros(N);
    (typeA,signA,valueA) = FPUnpack(addend_s, fpcr, fpexc);
    (type1,sign1,value1) = FPUnpack(op1_s, fpcr, fpexc);
    (type2,sign2,value2) = FPUnpack(op2_s, fpcr, fpexc);

    inf1 = (type1 == FPType_Infinity);
    inf2 = (type2 == FPType_Infinity);
    zero1 = (type1 == FPType_Zero);
    zero2 = (type2 == FPType_Zero);

    (done,result) = FPProcessNaNs3(typeA, type1, type2, addend_s, op1_s, op2_s, fpcr, fpexc);

    if !(HaveAltFP() && !UsingAArch32() && fpcr.AH == '1') then
        if typeA == FPType_QNaN && ((inf1 && zero2) || (zero1 && inf2)) then
            result = FPDefaultNaN(fpcr, 2*N);
            if fpexc then FPProcessException(FPExc_InvalidOp, fpcr);

    if !done then
        infA = (typeA == FPType_Infinity);
        zeroA = (typeA == FPType_Zero);

        // Determine sign and type product will have if it does not cause an
        // Invalid Operation.
        signP = sign1 EOR sign2;
        infP  = inf1 || inf2;
        zeroP = zero1 || zero2;

        // Non SNaN-generated Invalid Operation cases are multiplies of zero
        // by infinity and additions of opposite-signed infinities.
        invalidop = (inf1 && zero2) || (zero1 && inf2) || (infA && infP && signA != signP);

        if invalidop then
            result = FPDefaultNaN(fpcr, 2*N);
            if fpexc then FPProcessException(FPExc_InvalidOp, fpcr);

        // Other cases involving infinities produce an infinity of the same sign.
        elsif (infA && signA == '0') || (infP && signP == '0') then
            result = FPInfinity('0', 2*N);
        elsif (infA && signA == '1') || (infP && signP == '1') then
            result = FPInfinity('1', 2*N);

        // Cases where the result is exactly zero and its sign is not determined by the
        // rounding mode are additions of same-signed zeros.
        elsif zeroA && zeroP && signA == signP then
            result = FPZero(signA, 2*N);
```

```
        // Otherwise calculate numerical result and round it.
        else
            result_value = valueA + (value1 * value2);
            if result_value == 0.0 then  // Sign of exact zero result depends on rounding mode
                result_sign = if rounding == FPRounding_NEGINF then '1' else '0';
                result = FPZero(result_sign, 2*N);
            else
                result = FPRoundBF(result_value, fpcr, rounding, fpexc, 2*N);

        if !invalidop && fpexc then
            FPProcessDenorms3(typeA, type1, type2, 2*N, fpcr);

    return result<2*N-1:N>;
```

Library pseudocode for shared/functions/float/bfloat/BFMulAddH

```
// BFMulAddH()
// ===========
// Used by BFMLALB, BFMLALT, BFMLSLB and BFMLSLT instructions.

bits(N) BFMulAddH(bits(N) addend, bits(N DIV 2) op1, bits(N DIV 2) op2, FPCRType fpcr_in)
    assert N == 32;
    bits(N) value1 = op1 : Zeros(N DIV 2);
    bits(N) value2 = op2 : Zeros(N DIV 2);
    FPCRType fpcr = fpcr_in;
    boolean altfp = HaveAltFP() && fpcr.AH == '1';  // When TRUE:
    boolean fpexc = !altfp;                         //     Do not generate floating point exceptions
    if altfp then fpcr.<FIZ,FZ> = '11';             //     Flush denormal input and output to zero
    if altfp then fpcr.RMode    = '00';             //     Use RNE rounding mode
    return FPMulAdd(addend, value1, value2, fpcr, fpexc);
```

Library pseudocode for shared/functions/float/bfloat/BFMulAddH_ZA

```
// BFMulAddH_ZA()
// ==============
// Used by SME2 ZA-targeting BFMLAL and BFMLSL instructions.

bits(N) BFMulAddH_ZA(bits(N) addend, bits(N DIV 2) op1, bits(N DIV 2) op2, FPCRType fpcr)
    assert N == 32;
    bits(N) value1 = op1 : Zeros(N DIV 2);
    bits(N) value2 = op2 : Zeros(N DIV 2);
    return FPMulAdd_ZA(addend, value1, value2, fpcr);
```

Library pseudocode for shared/functions/float/bfloat/BFMulAdd_ZA

```
// BFMulAdd_ZA()
// =============
// Non-widening BFloat16 fused multiply-add used by SME2 ZA-targeting instructions.

bits(N) BFMulAdd_ZA(bits(N) addend, bits(N) op1, bits(N) op2, FPCRType fpcr_in)
    boolean fpexc = FALSE;
    FPCRType fpcr = fpcr_in;
    fpcr.DN = '1';          // Generate default NaN values
    return BFMulAdd(addend, op1, op2, fpcr, fpexc);
```

Library pseudocode for shared/functions/float/bfloat/BFMulH

```
// BFMulH()
// ========
// BFloat16 widening multiply to single-precision following BFloat16
// computation behaviors.

bits(2*N) BFMulH(bits(N) op1, bits(N) op2)

    assert N == 16;
    bits(2*N) result;

    FPCRType fpcr = FPCR[];
    (type1,sign1,value1) = BFUnpack(op1);
    (type2,sign2,value2) = BFUnpack(op2);
    if type1 == FPType_QNaN || type2 == FPType_QNaN then
        result = FPDefaultNaN(fpcr, 2*N);
    else
        inf1 = (type1 == FPType_Infinity);
        inf2 = (type2 == FPType_Infinity);
        zero1 = (type1 == FPType_Zero);
        zero2 = (type2 == FPType_Zero);
        if (inf1 && zero2) || (zero1 && inf2) then
            result = FPDefaultNaN(fpcr, 2*N);
        elsif inf1 || inf2 then
            result = FPInfinity(sign1 EOR sign2, 2*N);
        elsif zero1 || zero2 then
            result = FPZero(sign1 EOR sign2, 2*N);
        else
            result = BFRound(value1*value2, 2*N);

    return result;
```

Library pseudocode for shared/functions/float/bfloat/BFNeg

```
// BFNeg()
// =======

bits(N) BFNeg(bits(N) op)
    assert N == 16;
    boolean honor_altfp = TRUE;    // Honor alternate handling
    return BFNeg(op, honor_altfp);

// BFNeg()
// =======

bits(N) BFNeg(bits(N) op, boolean honor_altfp)

    assert N == 16;
    if honor_altfp && !UsingAArch32() && HaveAltFP() then
        FPCRType fpcr = FPCR[];
        if fpcr.AH == '1' then
            boolean fpexc = FALSE;
            boolean isbfloat16 = TRUE;
            (fptype, -, -) = FPUnpackBase(op, fpcr, fpexc, isbfloat16);
            if fptype IN {FPType_SNaN, FPType_QNaN} then

                return op;        // When fpcr.AH=1, sign of NaN has no consequence

    return NOT(op<N-1>) : op<N-2:0>;
```

Library pseudocode for shared/functions/float/bfloat/BFRound

```
// BFRound()
// =========
// Converts a real number OP into a single-precision value using the
// Round to Odd rounding mode and following BFloat16 computation behaviors.

bits(N) BFRound(real op, integer N)

    assert N == 32;
    assert op != 0.0;
    bits(N) result;

    // Format parameters - minimum exponent, numbers of exponent and fraction bits.
    minimum_exp = -126;  E = 8;  F = 23;

    // Split value into sign, unrounded mantissa and exponent.
    bit sign;
    real mantissa;
    if op < 0.0 then
        sign = '1';  mantissa = -op;
    else
        sign = '0';  mantissa = op;
    exponent = 0;
    while mantissa < 1.0 do
        mantissa = mantissa * 2.0;  exponent = exponent - 1;
    while mantissa >= 2.0 do
        mantissa = mantissa / 2.0;  exponent = exponent + 1;

    // Fixed Flush-to-zero.
    if exponent < minimum_exp then
        return FPZero(sign, N);

    // Start creating the exponent value for the result. Start by biasing the actual exponent
    // so that the minimum exponent becomes 1, lower values 0 (indicating possible underflow).
    biased_exp = Max((exponent - minimum_exp) + 1, 0);
    if biased_exp == 0 then mantissa = mantissa / 2.0^(minimum_exp - exponent);

    // Get the unrounded mantissa as an integer, and the "units in last place" rounding error.
    int_mant = RoundDown(mantissa * 2.0^F);  // < 2.0^F if biased_exp == 0, >= 2.0^F if not
    error = mantissa * 2.0^F - Real(int_mant);

    // Round to Odd
    if error != 0.0 then
        int_mant<0> = '1';

    // Deal with overflow and generate result.
    if biased_exp >= 2^E - 1 then
        result = FPInfinity(sign, N);      // Overflows generate appropriately-signed Infinity
    else
        result = sign : biased_exp<(N-2)-F:0> : int_mant<F-1:0>;

    return result;
```

Library pseudocode for shared/functions/float/bfloat/BFSub

```
// BFSub()
// =======
// Non-widening BFloat16 subtraction used by SVE2 instructions.

bits(N) BFSub(bits(N) op1, bits(N) op2, FPCRType fpcr)
    boolean fpexc = TRUE;
    return BFSub(op1, op2, fpcr, fpexc);

// BFSub()
// =======
// Non-widening BFloat16 subtraction following computational behaviors
// corresponding to instructions that read and write BFloat16 values.
// Calculates op1 - op2.
// The 'fpcr' argument supplies the FPCR control bits.

bits(N) BFSub(bits(N) op1, bits(N) op2, FPCRType fpcr, boolean fpexc)

    assert N == 16;
    FPRounding rounding = FPRoundingMode(fpcr);
    boolean done;
    bits(2*N) result;

    bits(2*N) op1_s = op1 : Zeros(N);
    bits(2*N) op2_s = op2 : Zeros(N);
    (type1,sign1,value1) = FPUnpack(op1_s, fpcr, fpexc);
    (type2,sign2,value2) = FPUnpack(op2_s, fpcr, fpexc);

    (done,result) = FPProcessNaNs(type1, type2, op1_s, op2_s, fpcr, fpexc);

    if !done then
        inf1 = (type1 == FPType_Infinity);
        inf2 = (type2 == FPType_Infinity);
        zero1 = (type1 == FPType_Zero);
        zero2 = (type2 == FPType_Zero);

        if inf1 && inf2 && sign1 == sign2 then
            result = FPDefaultNaN(fpcr, 2*N);
            if fpexc then FPProcessException(FPExc_InvalidOp, fpcr);
        elsif (inf1 && sign1 == '0') || (inf2 && sign2 == '1') then
            result = FPInfinity('0', 2*N);
        elsif (inf1 && sign1 == '1') || (inf2 && sign2 == '0') then
            result = FPInfinity('1', 2*N);
        elsif zero1 && zero2 && sign1 == NOT(sign2) then
            result = FPZero(sign1, 2*N);
        else
            result_value = value1 - value2;
            if result_value == 0.0 then  // Sign of exact zero result depends on rounding mode
                result_sign = if rounding == FPRounding_NEGINF then '1' else '0';
                result = FPZero(result_sign, 2*N);
            else
                result = FPRoundBF(result_value, fpcr, rounding, fpexc, 2*N);

        if fpexc then FPProcessDenorms(type1, type2, 2*N, fpcr);

    return result<2*N-1:N>;
```

Library pseudocode for shared/functions/float/bfloat/BFSub_ZA

```
// BFSub_ZA()
// ==========
// Non-widening BFloat16 subtraction used by SME2 ZA-targeting instructions.

bits(N) BFSub_ZA(bits(N) op1, bits(N) op2, FPCRType fpcr_in)
    boolean fpexc = FALSE;
    FPCRType fpcr = fpcr_in;
    fpcr.DN = '1';          // Generate default NaN values
    return BFSub(op1, op2, fpcr, fpexc);
```

Library pseudocode for shared/functions/float/bfloat/BFUnpack

```
// BFUnpack()
// ==========
// Unpacks a BFloat16 or single-precision value into its type,
// sign bit and real number that it represents.
// The real number result has the correct sign for numbers and infinities,
// is very large in magnitude for infinities, and is 0.0 for NaNs.
// (These values are chosen to simplify the description of
// comparisons and conversions.)

(FPType, bit, real) BFUnpack(bits(N) fpval)

    assert N IN {16,32};

    bit sign;
    bits(8) exp;
    bits(23) frac;
    if N == 16 then
        sign   = fpval<15>;
        exp    = fpval<14:7>;
        frac   = fpval<6:0> : Zeros(16);
    else  // N == 32
        sign   = fpval<31>;
        exp    = fpval<30:23>;
        frac   = fpval<22:0>;

    FPType fptype;
    real value;
    if IsZero(exp) then
        fptype = FPType_Zero;  value = 0.0;    // Fixed Flush to Zero
    elsif IsOnes(exp) then
        if IsZero(frac) then
            fptype = FPType_Infinity;  value = 2.0^1000000;
        else    // no SNaN for BF16 arithmetic
            fptype = FPType_QNaN; value = 0.0;
    else
        fptype = FPType_Nonzero;
        value = 2.0^(UInt(exp)-127) * (1.0 + Real(UInt(frac)) * 2.0^-23);

    if sign == '1' then value = -value;

    return (fptype, sign, value);
```

Library pseudocode for shared/functions/float/bfloat/BFZero

```
// BFZero()
// ========

bits(N) BFZero(bit sign, integer N)
    assert N == 16;
    constant integer E = 8;
    constant integer F = N - (E + 1);
    return sign : Zeros(E) : Zeros(F);
```

Library pseudocode for shared/functions/float/bfloat/FPAdd_BF16

```
// FPAdd_BF16()
// ============
// Single-precision add following BFloat16 computation behaviors.

bits(N) FPAdd_BF16(bits(N) op1, bits(N) op2)

    assert N == 32;
    bits(N) result;

    FPCRType fpcr = FPCR[];
    (type1,sign1,value1) = BFUnpack(op1);
    (type2,sign2,value2) = BFUnpack(op2);
    if type1 == FPType_QNaN || type2 == FPType_QNaN then
        result = FPDefaultNaN(fpcr, N);
    else
        inf1 = (type1 == FPType_Infinity);
        inf2 = (type2 == FPType_Infinity);
        zero1 = (type1 == FPType_Zero);
        zero2 = (type2 == FPType_Zero);
        if inf1 && inf2 && sign1 == NOT(sign2) then
            result = FPDefaultNaN(fpcr, N);
        elsif (inf1 && sign1 == '0') || (inf2 && sign2 == '0') then
            result = FPInfinity('0', N);
        elsif (inf1 && sign1 == '1') || (inf2 && sign2 == '1') then
            result = FPInfinity('1', N);
        elsif zero1 && zero2 && sign1 == sign2 then
            result = FPZero(sign1, N);
        else
            result_value = value1 + value2;
            if result_value == 0.0 then
                result = FPZero('0', N);    // Positive sign when Round to Odd
            else
                result = BFRound(result_value, N);

    return result;
```

Library pseudocode for shared/functions/float/bfloat/FPConvertBF

```
// FPConvertBF()
// =============
// Converts a single-precision OP to BFloat16 value using the
// Round to Nearest Even rounding mode when executed from AArch64 state and
// FPCR.AH == '1', otherwise rounding is controlled by FPCR/FPSCR.

bits(N DIV 2) FPConvertBF(bits(N) op, FPCRType fpcr_in, FPRounding rounding_in)

    assert N == 32;
    constant integer halfsize = N DIV 2;
    FPCRType fpcr = fpcr_in;
    FPRounding rounding = rounding_in;
    bits(N) result;                                // BF16 value in top 16 bits
    boolean altfp = HaveAltFP() && !UsingAArch32() && fpcr.AH == '1';
    boolean fpexc = !altfp;                         // Generate no floating-point exceptions
    if altfp then fpcr.<FIZ,FZ> = '11';             // Flush denormal input and output to zero
    if altfp then rounding = FPRounding_TIEEVEN;    // Use RNE rounding mode

    // Unpack floating-point operand, with always flush-to-zero if fpcr.AH == '1'.
    (fptype,sign,value) = FPUnpack(op, fpcr, fpexc);

    if fptype == FPType_SNaN || fptype == FPType_QNaN then
        if fpcr.DN == '1' then
            result = FPDefaultNaN(fpcr, N);
        else
            result = FPConvertNaN(op, N);
        if fptype == FPType_SNaN then
            if fpexc then FPProcessException(FPExc_InvalidOp, fpcr);
    elsif fptype == FPType_Infinity then
        result = FPInfinity(sign, N);
    elsif fptype == FPType_Zero then
        result = FPZero(sign, N);
    else
        result = FPRoundBF(value, fpcr, rounding, fpexc, N);

    // Returns correctly rounded BF16 value from top 16 bits
    return result<(2*halfsize)-1:halfsize>;

// FPConvertBF()
// =============
// Converts a single-precision operand to BFloat16 value.

bits(N DIV 2) FPConvertBF(bits(N) op, FPCRType fpcr)
    return FPConvertBF(op, fpcr, FPRoundingMode(fpcr));
```

Library pseudocode for shared/functions/float/bfloat/FPRoundBF

```
// FPRoundBF()
// ===========
// Converts a real number OP into a BFloat16 value using the supplied
// rounding mode RMODE. The 'fpexc' argument controls the generation of
// floating-point exceptions.

bits(N) FPRoundBF(real op, FPCRType fpcr, FPRounding rounding, boolean fpexc, integer N)
    assert N == 32;
    boolean isbfloat16 = TRUE;
    return FPRoundBase(op, fpcr, rounding, isbfloat16, fpexc, N);
```

Library pseudocode for shared/functions/float/fixedtofp/FixedToFP

```
// FixedToFP()
// ===========

// Convert M-bit fixed point 'op' with FBITS fractional bits to
// N-bit precision floating point, controlled by UNSIGNED and ROUNDING.

bits(N) FixedToFP(bits(M) op, integer fbits, boolean unsigned, FPCRType fpcr,
                  FPRounding rounding, integer N)

    assert N IN {16,32,64};
    assert M IN {16,32,64};
    bits(N) result;
    assert fbits >= 0;
    assert rounding != FPRounding_ODD;

    // Correct signed-ness
    int_operand = Int(op, unsigned);

    // Scale by fractional bits and generate a real value
    real_operand = Real(int_operand) / 2.0^fbits;

    if real_operand == 0.0 then
        result = FPZero('0', N);
    else
        result = FPRound(real_operand, fpcr, rounding, N);

    return result;
```

Library pseudocode for shared/functions/float/fpabs/FPAbs

```
// FPAbs()
// =======

bits(N) FPAbs(bits(N) op)

    assert N IN {16,32,64};
    if !UsingAArch32() && HaveAltFP() then
        FPCRType fpcr = FPCR[];
        if fpcr.AH == '1' then
            (fptype, -, -) = FPUnpack(op, fpcr, FALSE);
            if fptype IN {FPType_SNaN, FPType_QNaN} then
                return op;        // When fpcr.AH=1, sign of NaN has no consequence

    return '0' : op<N-2:0>;
```

Library pseudocode for shared/functions/float/fpadd/FPAdd

```
// FPAdd()
// =======

bits(N) FPAdd(bits(N) op1, bits(N) op2, FPCRType fpcr)
    boolean fpexc = TRUE;       // Generate floating-point exceptions
    return FPAdd(op1, op2, fpcr, fpexc);

// FPAdd()
// =======

bits(N) FPAdd(bits(N) op1, bits(N) op2, FPCRType fpcr, boolean fpexc)

    assert N IN {16,32,64};
    rounding = FPRoundingMode(fpcr);

    (type1,sign1,value1) = FPUnpack(op1, fpcr, fpexc);
    (type2,sign2,value2) = FPUnpack(op2, fpcr, fpexc);

    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr, fpexc);
    if !done then
        inf1  = (type1 == FPType_Infinity);  inf2  = (type2 == FPType_Infinity);
        zero1 = (type1 == FPType_Zero);      zero2 = (type2 == FPType_Zero);
        if inf1 && inf2 && sign1 == NOT(sign2) then
            result = FPDefaultNaN(fpcr, N);
            if fpexc then FPProcessException(FPExc_InvalidOp, fpcr);
        elsif (inf1 && sign1 == '0') || (inf2 && sign2 == '0') then
            result = FPInfinity('0', N);
        elsif (inf1 && sign1 == '1') || (inf2 && sign2 == '1') then
            result = FPInfinity('1', N);
        elsif zero1 && zero2 && sign1 == sign2 then
            result = FPZero(sign1, N);
        else
            result_value = value1 + value2;
            if result_value == 0.0 then  // Sign of exact zero result depends on rounding mode
                result_sign = if rounding == FPRounding_NEGINF then '1' else '0';
                result = FPZero(result_sign, N);
            else
                result = FPRound(result_value, fpcr, rounding, fpexc, N);

        if fpexc then FPProcessDenorms(type1, type2, N, fpcr);

    return result;
```

Library pseudocode for shared/functions/float/fpadd/FPAdd_ZA

```
// FPAdd_ZA()
// ==========
// Calculates op1+op2 for SME2 ZA-targeting instructions.

bits(N) FPAdd_ZA(bits(N) op1, bits(N) op2, FPCRType fpcr_in)
    FPCRType fpcr = fpcr_in;
    boolean fpexc = FALSE;      // Do not generate floating-point exceptions
    fpcr.DN = '1';              // Generate default NaN values
    return FPAdd(op1, op2, fpcr, fpexc);
```

Library pseudocode for shared/functions/float/fpcompare/FPCompare

```
// FPCompare()
// ===========

bits(4) FPCompare(bits(N) op1, bits(N) op2, boolean signal_nans, FPCRType fpcr)

    assert N IN {16,32,64};
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);

    bits(4) result;
    if type1 IN {FPType_SNaN, FPType_QNaN} || type2 IN {FPType_SNaN, FPType_QNaN} then
        result = '0011';
        if type1 == FPType_SNaN || type2 == FPType_SNaN || signal_nans then
            FPProcessException(FPExc_InvalidOp, fpcr);
    else
        // All non-NaN cases can be evaluated on the values produced by FPUnpack()
        if value1 == value2 then
            result = '0110';
        elsif value1 < value2 then
            result = '1000';
        else  // value1 > value2
            result = '0010';

        FPProcessDenorms(type1, type2, N, fpcr);

    return result;
```

Library pseudocode for shared/functions/float/fpcompareeq/FPCompareEQ

```
// FPCompareEQ()
// =============

boolean FPCompareEQ(bits(N) op1, bits(N) op2, FPCRType fpcr)

    assert N IN {16,32,64};
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);

    boolean result;
    if type1 IN {FPType_SNaN, FPType_QNaN} || type2 IN {FPType_SNaN, FPType_QNaN} then
        result = FALSE;
        if type1 == FPType_SNaN || type2 == FPType_SNaN then
            FPProcessException(FPExc_InvalidOp, fpcr);
    else
        // All non-NaN cases can be evaluated on the values produced by FPUnpack()
        result = (value1 == value2);
        FPProcessDenorms(type1, type2, N, fpcr);

    return result;
```

Library pseudocode for shared/functions/float/fpcomparege/FPCompareGE

```
// FPCompareGE()
// =============

boolean FPCompareGE(bits(N) op1, bits(N) op2, FPCRType fpcr)

    assert N IN {16,32,64};
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);

    boolean result;
    if type1 IN {FPType_SNaN, FPType_QNaN} || type2 IN {FPType_SNaN, FPType_QNaN} then
        result = FALSE;
        FPProcessException(FPExc_InvalidOp, fpcr);
    else
        // All non-NaN cases can be evaluated on the values produced by FPUnpack()
        result = (value1 >= value2);
        FPProcessDenorms(type1, type2, N, fpcr);

    return result;
```

Library pseudocode for shared/functions/float/fpcomparegt/FPCompareGT

```
// FPCompareGT()
// =============

boolean FPCompareGT(bits(N) op1, bits(N) op2, FPCRType fpcr)

    assert N IN {16,32,64};
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);

    boolean result;
    if type1 IN {FPType_SNaN, FPType_QNaN} || type2 IN {FPType_SNaN, FPType_QNaN} then
        result = FALSE;
        FPProcessException(FPExc_InvalidOp, fpcr);
    else
        // All non-NaN cases can be evaluated on the values produced by FPUnpack()
        result = (value1 > value2);
        FPProcessDenorms(type1, type2, N, fpcr);

    return result;
```

Library pseudocode for shared/functions/float/fpconvert/FPConvert

```
// FPConvert()
// ===========

// Convert floating point 'op' with N-bit precision to M-bit precision,
// with rounding controlled by ROUNDING.
// This is used by the FP-to-FP conversion instructions and so for
// half-precision data ignores FZ16, but observes AHP.

bits(M) FPConvert(bits(N) op, FPCRType fpcr, FPRounding rounding, integer M)

    assert M IN {16,32,64};
    assert N IN {16,32,64};
    bits(M) result;

    // Unpack floating-point operand optionally with flush-to-zero.
    (fptype,sign,value) = FPUnpackCV(op, fpcr);

    alt_hp = (M == 16) && (fpcr.AHP == '1');

    if fptype == FPType_SNaN || fptype == FPType_QNaN then
        if alt_hp then
            result = FPZero(sign, M);
        elsif fpcr.DN == '1' then
            result = FPDefaultNaN(fpcr, M);
        else
            result = FPConvertNaN(op, M);
        if fptype == FPType_SNaN || alt_hp then
            FPProcessException(FPExc_InvalidOp,fpcr);
    elsif fptype == FPType_Infinity then
        if alt_hp then
            result = sign:Ones(M-1);
            FPProcessException(FPExc_InvalidOp, fpcr);
        else
            result = FPInfinity(sign, M);
    elsif fptype == FPType_Zero then
        result = FPZero(sign, M);
    else
        result = FPRoundCV(value, fpcr, rounding, M);
        FPProcessDenorm(fptype, N, fpcr);

    return result;

// FPConvert()
// ===========

bits(M) FPConvert(bits(N) op, FPCRType fpcr, integer M)
    return FPConvert(op, fpcr, FPRoundingMode(fpcr), M);
```

Library pseudocode for shared/functions/float/fpconvertnan/FPConvertNaN

```
// FPConvertNaN()
// ==============
// Converts a NaN of one floating-point type to another

bits(M) FPConvertNaN(bits(N) op, integer M)

    assert N IN {16,32,64};
    assert M IN {16,32,64};
    bits(M) result;
    bits(51) frac;

    sign = op<N-1>;

    // Unpack payload from input NaN
    case N of
        when 64 frac = op<50:0>;
        when 32 frac = op<21:0>:Zeros(29);
        when 16 frac = op<8:0>:Zeros(42);

    // Repack payload into output NaN, while
    // converting an SNaN to a QNaN.
    case M of
        when 64 result = sign:Ones(M-52):frac;
        when 32 result = sign:Ones(M-23):frac<50:29>;
        when 16 result = sign:Ones(M-10):frac<50:42>;

    return result;
```

Library pseudocode for shared/functions/float/fpcrtype/FPCRType

```
type FPCRType;
```

Library pseudocode for shared/functions/float/fpdecoderm/FPDecodeRM

```
// FPDecodeRM()
// ============

// Decode most common AArch32 floating-point rounding encoding.

FPRounding FPDecodeRM(bits(2) rm)

    FPRounding result;
    case rm of
        when '00' result = FPRounding_TIEAWAY; // A
        when '01' result = FPRounding_TIEEVEN; // N
        when '10' result = FPRounding_POSINF;  // P
        when '11' result = FPRounding_NEGINF;  // M

    return result;
```

Library pseudocode for shared/functions/float/fpdecoderounding/FPDecodeRounding

```
// FPDecodeRounding()
// ==================

// Decode floating-point rounding mode and common AArch64 encoding.

FPRounding FPDecodeRounding(bits(2) rmode)
    case rmode of
        when '00' return FPRounding_TIEEVEN; // N
        when '01' return FPRounding_POSINF;  // P
        when '10' return FPRounding_NEGINF;  // M
        when '11' return FPRounding_ZERO;    // Z
```

Library pseudocode for shared/functions/float/fpdefaultnan/FPDefaultNaN

```
// FPDefaultNaN()
// ==============

bits(N) FPDefaultNaN(integer N)
    FPCRType fpcr = FPCR[];
    return FPDefaultNaN(fpcr, N);

bits(N) FPDefaultNaN(FPCRType fpcr, integer N)

    assert N IN {16,32,64};
    constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
    constant integer F = N - (E + 1);
    bit sign = if HaveAltFP() && !UsingAArch32() then fpcr.AH else '0';

    bits(E) exp  = Ones(E);
    bits(F) frac = '1':Zeros(F-1);

    return sign : exp : frac;
```

Library pseudocode for shared/functions/float/fpdiv/FPDiv

```
// FPDiv()
// =======

bits(N) FPDiv(bits(N) op1, bits(N) op2, FPCRType fpcr)

    assert N IN {16,32,64};
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);

    if !done then
        inf1  = type1 == FPType_Infinity;
        inf2  = type2 == FPType_Infinity;
        zero1 = type1 == FPType_Zero;
        zero2 = type2 == FPType_Zero;

        if (inf1 && inf2) || (zero1 && zero2) then
            result = FPDefaultNaN(fpcr, N);
            FPProcessException(FPExc_InvalidOp, fpcr);
        elsif inf1 || zero2 then
            result = FPInfinity(sign1 EOR sign2, N);
            if !inf1 then FPProcessException(FPExc_DivideByZero, fpcr);
        elsif zero1 || inf2 then
            result = FPZero(sign1 EOR sign2, N);
        else
            result = FPRound(value1/value2, fpcr, N);

        if !zero2 then
            FPProcessDenorms(type1, type2, N, fpcr);

    return result;
```

Library pseudocode for shared/functions/float/fpdot/FPDot

```
// FPDot()
// =======
// Calculates single-precision result of 2-way 16-bit floating-point dot-product
// with a single rounding.
// The 'fpcr' argument supplies the FPCR control bits and 'isbfloat16'
// determines whether input operands are BFloat16 or half-precision type.
// and 'fpexc' controls the generation of floating-point exceptions.

bits(N) FPDot(bits(N DIV 2) op1_a, bits(N DIV 2) op1_b, bits(N DIV 2) op2_a,
              bits(N DIV 2) op2_b, FPCRType fpcr, boolean isbfloat16)
    boolean fpexc = TRUE;       // Generate floating-point exceptions
    return FPDot(op1_a, op1_b, op2_a, op2_b, fpcr, isbfloat16, fpexc);

bits(N) FPDot(bits(N DIV 2) op1_a, bits(N DIV 2) op1_b, bits(N DIV 2) op2_a,
              bits(N DIV 2) op2_b, FPCRType fpcr_in, boolean isbfloat16, boolean fpexc)
    FPCRType fpcr = fpcr_in;

    assert N == 32;
    bits(N) result;
    boolean done;
    fpcr.AHP = '0';           // Ignore alternative half-precision option
    rounding = FPRoundingMode(fpcr);

    (type1_a,sign1_a,value1_a) = FPUnpackBase(op1_a, fpcr, fpexc, isbfloat16);
    (type1_b,sign1_b,value1_b) = FPUnpackBase(op1_b, fpcr, fpexc, isbfloat16);
    (type2_a,sign2_a,value2_a) = FPUnpackBase(op2_a, fpcr, fpexc, isbfloat16);
    (type2_b,sign2_b,value2_b) = FPUnpackBase(op2_b, fpcr, fpexc, isbfloat16);

    inf1_a = (type1_a == FPType_Infinity); zero1_a = (type1_a == FPType_Zero);
    inf1_b = (type1_b == FPType_Infinity); zero1_b = (type1_b == FPType_Zero);
    inf2_a = (type2_a == FPType_Infinity); zero2_a = (type2_a == FPType_Zero);
    inf2_b = (type2_b == FPType_Infinity); zero2_b = (type2_b == FPType_Zero);

    (done,result) = FPProcessNaNs4(type1_a, type1_b, type2_a, type2_b,
                                   op1_a, op1_b, op2_a, op2_b, fpcr, fpexc);

    if (((inf1_a && zero2_a) || (zero1_a && inf2_a)) &&
        ((inf1_b && zero2_b) || (zero1_b && inf2_b))) then
        result = FPDefaultNaN(fpcr, N);
        if fpexc then FPProcessException(FPExc_InvalidOp, fpcr);

    if !done then
        // Determine sign and type products will have if it does not cause an Invalid
        // Operation.
        signPa = sign1_a EOR sign2_a;
        signPb = sign1_b EOR sign2_b;
        infPa = inf1_a || inf2_a;
        infPb = inf1_b || inf2_b;
        zeroPa = zero1_a || zero2_a;
        zeroPb = zero1_b || zero2_b;

        // Non SNaN-generated Invalid Operation cases are multiplies of zero
        // by infinity and additions of opposite-signed infinities.
        invalidop = ((inf1_a && zero2_a) || (zero1_a && inf2_a) ||
            (inf1_b && zero2_b) || (zero1_b && inf2_b) || (infPa && infPb && signPa != signPb));

        if invalidop then
            result = FPDefaultNaN(fpcr, N);
            if fpexc then FPProcessException(FPExc_InvalidOp, fpcr);

       // Other cases involving infinities produce an infinity of the same sign.
        elsif (infPa && signPa == '0') || (infPb && signPb == '0') then
            result = FPInfinity('0', N);
        elsif (infPa && signPa == '1') || (infPb && signPb == '1') then
            result = FPInfinity('1', N);

        // Cases where the result is exactly zero and its sign is not determined by the
        // rounding mode are additions of same-signed zeros.
        elsif zeroPa && zeroPb && signPa == signPb then
            result = FPZero(signPa, N);
```

```
        // Otherwise calculate fused sum of products and round it.
        else
            result_value = (value1_a * value2_a) + (value1_b * value2_b);
            if result_value == 0.0 then  // Sign of exact zero result depends on rounding mode
                result_sign = if rounding == FPRounding_NEGINF then '1' else '0';
                result = FPZero(result_sign, N);
            else
                result = FPRound(result_value, fpcr, rounding, fpexc, N);

    return result;
```

Library pseudocode for shared/functions/float/fpdot/FPDotAdd

```
// FPDotAdd()
// ==========
// Half-precision 2-way dot-product and add to single-precision.

bits(N) FPDotAdd(bits(N) addend, bits(N DIV 2) op1_a, bits(N DIV 2) op1_b,
                 bits(N DIV 2) op2_a, bits(N DIV 2) op2_b, FPCRType fpcr)
    assert N == 32;

    bits(N) prod;
    boolean isbfloat16 = FALSE;
    boolean fpexc = TRUE;     // Generate floating-point exceptions
    prod = FPDot(op1_a, op1_b, op2_a, op2_b, fpcr, isbfloat16, fpexc);
    result = FPAdd(addend, prod, fpcr, fpexc);

    return result;
```

Library pseudocode for shared/functions/float/fpdot/FPDotAdd_ZA

```
// FPDotAdd_ZA()
// =============
// Half-precision 2-way dot-product and add to single-precision
// for SME ZA-targeting instructions.

bits(N) FPDotAdd_ZA(bits(N) addend, bits(N DIV 2) op1_a, bits(N DIV 2) op1_b,
                    bits(N DIV 2) op2_a, bits(N DIV 2) op2_b, FPCRType fpcr_in)
    FPCRType fpcr = fpcr_in;
    assert N == 32;

    bits(N) prod;
    boolean isbfloat16 = FALSE;
    boolean fpexc = FALSE;    // Do not generate floating-point exceptions
    fpcr.DN = '1';            // Generate default NaN values
    prod = FPDot(op1_a, op1_b, op2_a, op2_b, fpcr, isbfloat16, fpexc);
    result = FPAdd(addend, prod, fpcr, fpexc);

    return result;
```

Library pseudocode for shared/functions/float/fpexc/FPExc

```
// FPExc
// =====

enumeration FPExc       {FPExc_InvalidOp, FPExc_DivideByZero, FPExc_Overflow,
                         FPExc_Underflow, FPExc_Inexact, FPExc_InputDenorm};
```

Library pseudocode for shared/functions/float/fpinfinity/FPInfinity

```
// FPInfinity()
// ============

bits(N) FPInfinity(bit sign, integer N)

    assert N IN {16,32,64};
    constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
    constant integer F = N - (E + 1);
    bits(E) exp  = Ones(E);
    bits(F) frac = Zeros(F);

    return sign : exp : frac;
```

Library pseudocode for shared/functions/float/fpmatmul/FPMatMulAdd

```
// FPMatMulAdd()
// =============
//
// Floating point matrix multiply and add to same precision matrix
// result[2, 2] = addend[2, 2] + (op1[2, 2] * op2[2, 2])

bits(N) FPMatMulAdd(bits(N) addend, bits(N) op1, bits(N) op2, integer esize, FPCRType fpcr)

    assert N == esize * 2 * 2;
    bits(N)  result;
    bits(esize) prod0, prod1, sum;

    for i = 0 to 1
        for j = 0 to 1
            sum   = Elem[addend, 2*i + j, esize];
            prod0 = FPMul(Elem[op1, 2*i + 0, esize],
                          Elem[op2, 2*j + 0, esize], fpcr);
            prod1 = FPMul(Elem[op1, 2*i + 1, esize],
                          Elem[op2, 2*j + 1, esize], fpcr);
            sum   = FPAdd(sum, FPAdd(prod0, prod1, fpcr), fpcr);
            Elem[result, 2*i + j, esize] = sum;

    return result;
```

Library pseudocode for shared/functions/float/fpmax/FPMax

```
// FPMax()
// =======

bits(N) FPMax(bits(N) op1, bits(N) op2, FPCRType fpcr)
    boolean altfp = HaveAltFP() && !UsingAArch32() && fpcr.AH == '1';
    return FPMax(op1, op2, fpcr, altfp);

// FPMax()
// =======
// Compare two inputs and return the larger value after rounding. The
// 'fpcr' argument supplies the FPCR control bits and 'altfp' determines
// if the function should use alternative floating-point behavior.

bits(N) FPMax(bits(N) op1, bits(N) op2, FPCRType fpcr_in, boolean altfp)

    assert N IN {16,32,64};
    boolean done;
    bits(N) result;
    FPCRType fpcr = fpcr_in;
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);

    if altfp && type1 == FPType_Zero && type2 == FPType_Zero && sign1 != sign2 then
        // Alternate handling of zeros with differing sign
        return FPZero(sign2, N);
    elsif altfp && (type1 IN {FPType_SNaN, FPType_QNaN} || type2 IN {FPType_SNaN, FPType_QNaN}) then
        // Alternate handling of NaN inputs
        FPProcessException(FPExc_InvalidOp, fpcr);
        return (if type2 == FPType_Zero then FPZero(sign2, N) else op2);

    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);
    if !done then
        FPType fptype;
        bit sign;
        real value;
        if value1 > value2 then
            (fptype,sign,value) = (type1,sign1,value1);
        else
            (fptype,sign,value) = (type2,sign2,value2);
        if fptype == FPType_Infinity then
            result = FPInfinity(sign, N);
        elsif fptype == FPType_Zero then
            sign = sign1 AND sign2;         // Use most positive sign
            result = FPZero(sign, N);
        else
            // The use of FPRound() covers the case where there is a trapped underflow exception
            // for a denormalized number even though the result is exact.
            rounding = FPRoundingMode(fpcr);
            if altfp then    // Denormal output is not flushed to zero
                fpcr.FZ = '0';
                fpcr.FZ16 = '0';

            result = FPRound(value, fpcr, rounding, TRUE, N);

        FPProcessDenorms(type1, type2, N, fpcr);

    return result;
```

Library pseudocode for shared/functions/float/fpmaxnormal/FPMaxNormal

```
// FPMaxNormal()
// =============

bits(N) FPMaxNormal(bit sign, integer N)

    assert N IN {16,32,64};
    constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
    constant integer F = N - (E + 1);
    exp  = Ones(E-1):'0';
    frac = Ones(F);

    return sign : exp : frac;
```

Library pseudocode for shared/functions/float/fpmaxnum/FPMaxNum

```
// FPMaxNum()
// ==========

bits(N) FPMaxNum(bits(N) op1_in, bits(N) op2_in, FPCRType fpcr)

    assert N IN {16,32,64};
    bits(N) op1 = op1_in;
    bits(N) op2 = op2_in;
    (type1,-,-) = FPUnpack(op1, fpcr);
    (type2,-,-) = FPUnpack(op2, fpcr);

    boolean type1_nan = type1 IN {FPType_QNaN, FPType_SNaN};
    boolean type2_nan = type2 IN {FPType_QNaN, FPType_SNaN};
    boolean altfp = HaveAltFP() && !UsingAArch32() && fpcr.AH == '1';

    if !(altfp && type1_nan && type2_nan) then
        // Treat a single quiet-NaN as -Infinity.
        if type1 == FPType_QNaN && type2 != FPType_QNaN then
            op1 = FPInfinity('1', N);
        elsif type1 != FPType_QNaN && type2 == FPType_QNaN then
            op2 = FPInfinity('1', N);

    altfmaxfmin = FALSE;    // Restrict use of FMAX/FMIN NaN propagation rules
    result = FPMax(op1, op2, fpcr, altfmaxfmin);

    return result;
```

Library pseudocode for shared/functions/float/fpmerge/IsMerging

```
// IsMerging()
// ===========
// Returns TRUE if the output elements other than the lowest are taken from
// the destination register.

boolean IsMerging(FPCRType fpcr)
    bit nep = if HaveSME() && PSTATE.SM == '1' && !IsFullA64Enabled() then '0' else fpcr.NEP;
    return HaveAltFP() && !UsingAArch32() && nep == '1';
```

Library pseudocode for shared/functions/float/fpmin/FPMin

```
// FPMin()
// =======

bits(N) FPMin(bits(N) op1, bits(N) op2, FPCRType fpcr)
    boolean altfp = HaveAltFP() && !UsingAArch32() && fpcr.AH == '1';
    return FPMin(op1, op2, fpcr, altfp);

// FPMin()
// =======
// Compare two operands and return the smaller operand after rounding. The
// 'fpcr' argument supplies the FPCR control bits and 'altfp' determines
// if the function should use alternative behavior.

bits(N) FPMin(bits(N) op1, bits(N) op2, FPCRType fpcr_in, boolean altfp)

    assert N IN {16,32,64};
    boolean done;
    bits(N) result;
    FPCRType fpcr = fpcr_in;
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);

    if altfp && type1 == FPType_Zero && type2 == FPType_Zero && sign1 != sign2 then
        // Alternate handling of zeros with differing sign
        return FPZero(sign2, N);
    elsif altfp && (type1 IN {FPType_SNaN, FPType_QNaN} || type2 IN {FPType_SNaN, FPType_QNaN}) then
        // Alternate handling of NaN inputs
        FPProcessException(FPExc_InvalidOp, fpcr);
        return (if type2 == FPType_Zero then FPZero(sign2, N) else op2);

    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);
    if !done then
        FPType fptype;
        bit sign;
        real value;
        FPRounding rounding;
        if value1 < value2 then
            (fptype,sign,value) = (type1,sign1,value1);
        else
            (fptype,sign,value) = (type2,sign2,value2);
        if fptype == FPType_Infinity then
            result = FPInfinity(sign, N);
        elsif fptype == FPType_Zero then
            sign = sign1 OR sign2;              // Use most negative sign
            result = FPZero(sign, N);
        else
            // The use of FPRound() covers the case where there is a trapped underflow exception
            // for a denormalized number even though the result is exact.
            rounding = FPRoundingMode(fpcr);
            if altfp then    // Denormal output is not flushed to zero
                fpcr.FZ = '0';
                fpcr.FZ16 = '0';

            result = FPRound(value, fpcr, rounding, TRUE, N);

        FPProcessDenorms(type1, type2, N, fpcr);

    return result;
```

Library pseudocode for shared/functions/float/fpminnum/FPMinNum

```
// FPMinNum()
// ==========

bits(N) FPMinNum(bits(N) op1_in, bits(N) op2_in, FPCRType fpcr)

    assert N IN {16,32,64};
    bits(N) op1 = op1_in;
    bits(N) op2 = op2_in;
    (type1,-,-) = FPUnpack(op1, fpcr);
    (type2,-,-) = FPUnpack(op2, fpcr);

    boolean type1_nan = type1 IN {FPType_QNaN, FPType_SNaN};
    boolean type2_nan = type2 IN {FPType_QNaN, FPType_SNaN};
    boolean altfp = HaveAltFP() && !UsingAArch32() && fpcr.AH == '1';

    if !(altfp && type1_nan && type2_nan) then
        // Treat a single quiet-NaN as +Infinity.
        if type1 == FPType_QNaN && type2 != FPType_QNaN then
            op1 = FPInfinity('0', N);
        elsif type1 != FPType_QNaN && type2 == FPType_QNaN then
            op2 = FPInfinity('0', N);

    altfmaxfmin = FALSE;    // Restrict use of FMAX/FMIN NaN propagation rules
    result = FPMin(op1, op2, fpcr, altfmaxfmin);

    return result;
```

Library pseudocode for shared/functions/float/fpmul/FPMul

```
// FPMul()
// =======

bits(N) FPMul(bits(N) op1, bits(N) op2, FPCRType fpcr)

    assert N IN {16,32,64};
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);
    if !done then
        inf1 = (type1 == FPType_Infinity);
        inf2 = (type2 == FPType_Infinity);
        zero1 = (type1 == FPType_Zero);
        zero2 = (type2 == FPType_Zero);

        if (inf1 && zero2) || (zero1 && inf2) then
            result = FPDefaultNaN(fpcr, N);
            FPProcessException(FPExc_InvalidOp, fpcr);
        elsif inf1 || inf2 then
            result = FPInfinity(sign1 EOR sign2, N);
        elsif zero1 || zero2 then
            result = FPZero(sign1 EOR sign2, N);
        else
            result = FPRound(value1*value2, fpcr, N);

        FPProcessDenorms(type1, type2, N, fpcr);

    return result;
```

Library pseudocode for shared/functions/float/fpmuladd/FPMulAdd

```
// FPMulAdd()
// ==========

bits(N) FPMulAdd(bits(N) addend, bits(N) op1, bits(N) op2, FPCRType fpcr)
    boolean fpexc = TRUE;       // Generate floating-point exceptions
    return FPMulAdd(addend, op1, op2, fpcr, fpexc);

// FPMulAdd()
// ==========
//
// Calculates addend + op1*op2 with a single rounding. The 'fpcr' argument
// supplies the FPCR control bits, and 'fpexc' controls the generation of
// floating-point exceptions.

bits(N) FPMulAdd(bits(N) addend, bits(N) op1, bits(N) op2,
                 FPCRType fpcr, boolean fpexc)

    assert N IN {16,32,64};

    (typeA,signA,valueA) = FPUnpack(addend, fpcr, fpexc);
    (type1,sign1,value1) = FPUnpack(op1, fpcr, fpexc);
    (type2,sign2,value2) = FPUnpack(op2, fpcr, fpexc);
    rounding = FPRoundingMode(fpcr);
    inf1 = (type1 == FPType_Infinity); zero1 = (type1 == FPType_Zero);
    inf2 = (type2 == FPType_Infinity); zero2 = (type2 == FPType_Zero);

    (done,result) = FPProcessNaNs3(typeA, type1, type2, addend, op1, op2, fpcr, fpexc);

    if !(HaveAltFP() && !UsingAArch32() && fpcr.AH == '1') then
        if typeA == FPType_QNaN && ((inf1 && zero2) || (zero1 && inf2)) then
            result = FPDefaultNaN(fpcr, N);
            if fpexc then FPProcessException(FPExc_InvalidOp, fpcr);

    if !done then
        infA = (typeA == FPType_Infinity); zeroA = (typeA == FPType_Zero);

        // Determine sign and type product will have if it does not cause an
        // Invalid Operation.
        signP = sign1 EOR sign2;
        infP  = inf1 || inf2;
        zeroP = zero1 || zero2;

        // Non SNaN-generated Invalid Operation cases are multiplies of zero
        // by infinity and additions of opposite-signed infinities.
        invalidop = (inf1 && zero2) || (zero1 && inf2) || (infA && infP && signA != signP);

        if invalidop then
            result = FPDefaultNaN(fpcr, N);
            if fpexc then FPProcessException(FPExc_InvalidOp, fpcr);
        // Other cases involving infinities produce an infinity of the same sign.
        elsif (infA && signA == '0') || (infP && signP == '0') then
            result = FPInfinity('0', N);
        elsif (infA && signA == '1') || (infP && signP == '1') then
            result = FPInfinity('1', N);

        // Cases where the result is exactly zero and its sign is not determined by the
        // rounding mode are additions of same-signed zeros.
        elsif zeroA && zeroP && signA == signP then
            result = FPZero(signA, N);

        // Otherwise calculate numerical result and round it.
        else
            result_value = valueA + (value1 * value2);
            if result_value == 0.0 then  // Sign of exact zero result depends on rounding mode
                result_sign = if rounding == FPRounding_NEGINF then '1' else '0';
                result = FPZero(result_sign, N);
            else
                result = FPRound(result_value, fpcr, rounding, fpexc, N);

        if !invalidop && fpexc then
```

```
            FPProcessDenorms3(typeA, type1, type2, N, fpcr);

    return result;
```

Library pseudocode for shared/functions/float/fpmuladd/FPMulAdd_ZA

```
// FPMulAdd_ZA()
// =============
// Calculates addend + op1*op2 with a single rounding for SME ZA-targeting
// instructions.

bits(N) FPMulAdd_ZA(bits(N) addend, bits(N) op1, bits(N) op2, FPCRType fpcr_in)
    FPCRType fpcr = fpcr_in;
    boolean fpexc = FALSE;      // Do not generate floating-point exceptions
    fpcr.DN = '1';              // Generate default NaN values
    return FPMulAdd(addend, op1, op2, fpcr, fpexc);
```

Library pseudocode for shared/functions/float/fpmuladdh/FPMulAddH

```
// FPMulAddH()
// ===========
// Calculates addend + op1*op2.

bits(N) FPMulAddH(bits(N) addend, bits(N DIV 2) op1, bits(N DIV 2) op2, FPCRType fpcr)
    boolean fpexc = TRUE;       // Generate floating-point exceptions
    return FPMulAddH(addend, op1, op2, fpcr, fpexc);

// FPMulAddH()
// ===========
// Calculates addend + op1*op2.

bits(N) FPMulAddH(bits(N) addend, bits(N DIV 2) op1, bits(N DIV 2) op2,
                  FPCRType fpcr, boolean fpexc)

    assert N == 32;
    rounding = FPRoundingMode(fpcr);
    (typeA,signA,valueA) = FPUnpack(addend, fpcr, fpexc);
    (type1,sign1,value1) = FPUnpack(op1, fpcr, fpexc);
    (type2,sign2,value2) = FPUnpack(op2, fpcr, fpexc);
    inf1 = (type1 == FPType_Infinity); zero1 = (type1 == FPType_Zero);
    inf2 = (type2 == FPType_Infinity); zero2 = (type2 == FPType_Zero);

    (done,result) = FPProcessNaNs3H(typeA, type1, type2, addend, op1, op2, fpcr, fpexc);

    if !(HaveAltFP() && !UsingAArch32() && fpcr.AH == '1') then
        if typeA == FPType_QNaN && ((inf1 && zero2) || (zero1 && inf2)) then
            result = FPDefaultNaN(fpcr, N);
            if fpexc then FPProcessException(FPExc_InvalidOp, fpcr);

    if !done then
        infA = (typeA == FPType_Infinity); zeroA = (typeA == FPType_Zero);

        // Determine sign and type product will have if it does not cause an
        // Invalid Operation.
        signP = sign1 EOR sign2;
        infP = inf1 || inf2;
        zeroP = zero1 || zero2;

        // Non SNaN-generated Invalid Operation cases are multiplies of zero by infinity and
        // additions of opposite-signed infinities.
        invalidop = (inf1 && zero2) || (zero1 && inf2) || (infA && infP && signA != signP);

        if invalidop then
            result = FPDefaultNaN(fpcr, N);
            if fpexc then FPProcessException(FPExc_InvalidOp, fpcr);

        // Other cases involving infinities produce an infinity of the same sign.
        elsif (infA && signA == '0') || (infP && signP == '0') then
            result = FPInfinity('0', N);
        elsif (infA && signA == '1') || (infP && signP == '1') then
            result = FPInfinity('1', N);

        // Cases where the result is exactly zero and its sign is not determined by the
        // rounding mode are additions of same-signed zeros.
        elsif zeroA && zeroP && signA == signP then
            result = FPZero(signA, N);

        // Otherwise calculate numerical result and round it.
        else
            result_value = valueA + (value1 * value2);
            if result_value == 0.0 then // Sign of exact zero result depends on rounding mode
                result_sign = if rounding == FPRounding_NEGINF then '1' else '0';
                result = FPZero(result_sign, N);
            else
                result = FPRound(result_value, fpcr, rounding, fpexc, N);

        if !invalidop && fpexc then
            FPProcessDenorm(typeA, N, fpcr);
```

```
    return result;
```

Library pseudocode for shared/functions/float/fpmuladdh/FPMulAddH_ZA

```
// FPMulAddH_ZA()
// ==============
// Calculates addend + op1*op2 for SME2 ZA-targeting instructions.

bits(N) FPMulAddH_ZA(bits(N) addend, bits(N DIV 2) op1, bits(N DIV 2) op2, FPCRType fpcr_in)
    FPCRType fpcr = fpcr_in;
    boolean fpexc = FALSE;      // Do not generate floating-point exceptions
    fpcr.DN = '1';              // Generate default NaN values
    return FPMulAddH(addend, op1, op2, fpcr, fpexc);
```

Library pseudocode for shared/functions/float/fpmuladdh/FPProcessNaNs3H

```
// FPProcessNaNs3H()
// =================

(boolean, bits(N)) FPProcessNaNs3H(FPType type1, FPType type2, FPType type3,
                                   bits(N) op1, bits(N DIV 2) op2, bits(N DIV 2) op3,
                                   FPCRType fpcr, boolean fpexc)

    assert N IN {32,64};

    bits(N) result;
    FPType type_nan;
    // When TRUE, use alternative NaN propagation rules.
    boolean altfp   = HaveAltFP() && !UsingAArch32() && fpcr.AH == '1';
    boolean op1_nan = type1 IN {FPType_SNaN, FPType_QNaN};
    boolean op2_nan = type2 IN {FPType_SNaN, FPType_QNaN};
    boolean op3_nan = type3 IN {FPType_SNaN, FPType_QNaN};
    if altfp then
        if (type1 == FPType_SNaN || type2 == FPType_SNaN || type3 == FPType_SNaN) then
            type_nan = FPType_SNaN;
        else
            type_nan = FPType_QNaN;

    boolean done;
    if altfp && op1_nan && op2_nan && op3_nan then          // <n> register NaN selected
        done = TRUE;  result = FPConvertNaN(FPProcessNaN(type_nan, op2, fpcr, fpexc), N);
    elsif altfp && op2_nan && (op1_nan || op3_nan) then     // <n> register NaN selected
        done = TRUE;  result = FPConvertNaN(FPProcessNaN(type_nan, op2, fpcr, fpexc), N);
    elsif altfp && op3_nan && op1_nan then                  // <m> register NaN selected
        done = TRUE;  result = FPConvertNaN(FPProcessNaN(type_nan, op3, fpcr, fpexc), N);
    elsif type1 == FPType_SNaN then
        done = TRUE; result = FPProcessNaN(type1, op1, fpcr, fpexc);
    elsif type2 == FPType_SNaN then
        done = TRUE; result = FPConvertNaN(FPProcessNaN(type2, op2, fpcr, fpexc), N);
    elsif type3 == FPType_SNaN then
        done = TRUE; result = FPConvertNaN(FPProcessNaN(type3, op3, fpcr, fpexc), N);
    elsif type1 == FPType_QNaN then
        done = TRUE; result = FPProcessNaN(type1, op1, fpcr, fpexc);
    elsif type2 == FPType_QNaN then
        done = TRUE; result = FPConvertNaN(FPProcessNaN(type2, op2, fpcr, fpexc), N);
    elsif type3 == FPType_QNaN then
        done = TRUE; result = FPConvertNaN(FPProcessNaN(type3, op3, fpcr, fpexc), N);
    else
        done = FALSE; result = Zeros(N); // 'Don't care' result
    return (done, result);
```

Library pseudocode for shared/functions/float/fpmulx/FPMulX

```
// FPMulX()
// ========

bits(N) FPMulX(bits(N) op1, bits(N) op2, FPCRType fpcr)

    assert N IN {16,32,64};
    bits(N) result;
    boolean done;
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);

    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);
    if !done then
        inf1 = (type1 == FPType_Infinity);
        inf2 = (type2 == FPType_Infinity);
        zero1 = (type1 == FPType_Zero);
        zero2 = (type2 == FPType_Zero);

        if (inf1 && zero2) || (zero1 && inf2) then
            result = FPTwo(sign1 EOR sign2, N);
        elsif inf1 || inf2 then
            result = FPInfinity(sign1 EOR sign2, N);
        elsif zero1 || zero2 then
            result = FPZero(sign1 EOR sign2, N);
        else
            result = FPRound(value1*value2, fpcr, N);

        FPProcessDenorms(type1, type2, N, fpcr);

    return result;
```

Library pseudocode for shared/functions/float/fpneg/FPNeg

```
// FPNeg()
// =======

bits(N) FPNeg(bits(N) op)

    assert N IN {16,32,64};
    if !UsingAArch32() && HaveAltFP() then
        FPCRType fpcr = FPCR[];
        if fpcr.AH == '1' then
            (fptype, -, -) = FPUnpack(op, fpcr, FALSE);
            if fptype IN {FPType_SNaN, FPType_QNaN} then

                return op;        // When fpcr.AH=1, sign of NaN has no consequence

    return NOT(op<N-1>) : op<N-2:0>;
```

Library pseudocode for shared/functions/float/fponepointfive/FPOnePointFive

```
// FPOnePointFive()
// ================

bits(N) FPOnePointFive(bit sign, integer N)

    assert N IN {16,32,64};
    constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
    constant integer F = N - (E + 1);
    exp  = '0':Ones(E-1);
    frac = '1':Zeros(F-1);
    result = sign : exp : frac;

    return result;
```

Library pseudocode for shared/functions/float/fpprocessdenorms/FPProcessDenorm

```
// FPProcessDenorm()
// =================
// Handles denormal input in case of single-precision or double-precision
// when using alternative floating-point mode.

FPProcessDenorm(FPType fptype, integer N, FPCRType fpcr)
    boolean altfp = HaveAltFP() && !UsingAArch32() && fpcr.AH == '1';
    if altfp && N != 16 && fptype == FPType_Denormal then
        FPProcessException(FPExc_InputDenorm, fpcr);
```

Library pseudocode for shared/functions/float/fpprocessdenorms/FPProcessDenorms

```
// FPProcessDenorms()
// ==================
// Handles denormal input in case of single-precision or double-precision
// when using alternative floating-point mode.

FPProcessDenorms(FPType type1, FPType type2, integer N, FPCRType fpcr)
    boolean altfp = HaveAltFP() && !UsingAArch32() && fpcr.AH == '1';
    if altfp && N != 16 && (type1 == FPType_Denormal || type2 == FPType_Denormal) then
        FPProcessException(FPExc_InputDenorm, fpcr);
```

Library pseudocode for shared/functions/float/fpprocessdenorms/FPProcessDenorms3

```
// FPProcessDenorms3()
// ===================
// Handles denormal input in case of single-precision or double-precision
// when using alternative floating-point mode.

FPProcessDenorms3(FPType type1, FPType type2, FPType type3, integer N, FPCRType fpcr)
    boolean altfp = HaveAltFP() && !UsingAArch32() && fpcr.AH == '1';
    if altfp && N != 16 && (type1 == FPType_Denormal || type2 == FPType_Denormal ||
        type3 == FPType_Denormal) then
        FPProcessException(FPExc_InputDenorm, fpcr);
```

Library pseudocode for shared/functions/float/fpprocessdenorms/FPProcessDenorms4

```
// FPProcessDenorms4()
// ===================
// Handles denormal input in case of single-precision or double-precision
// when using alternative floating-point mode.

FPProcessDenorms4(FPType type1, FPType type2, FPType type3, FPType type4, integer N, FPCRType fpcr)
    boolean altfp = HaveAltFP() && !UsingAArch32() && fpcr.AH == '1';
    if altfp && N != 16 && (type1 == FPType_Denormal || type2 == FPType_Denormal ||
        type3 == FPType_Denormal || type4 == FPType_Denormal) then
        FPProcessException(FPExc_InputDenorm, fpcr);
```

Library pseudocode for shared/functions/float/fpprocessexception/FPProcessException

```
// FPProcessException()
// ====================
//
// The 'fpcr' argument supplies FPCR control bits. Status information is
// updated directly in the FPSR where appropriate.

FPProcessException(FPExc except, FPCRType fpcr)

    integer cumul;
    // Determine the cumulative exception bit number
    case except of
        when FPExc_InvalidOp     cumul = 0;
        when FPExc_DivideByZero  cumul = 1;
        when FPExc_Overflow      cumul = 2;
        when FPExc_Underflow     cumul = 3;
        when FPExc_Inexact       cumul = 4;
        when FPExc_InputDenorm   cumul = 7;
    enable = cumul + 8;
    if fpcr<enable> == '1' && (!HaveSME() || PSTATE.SM == '0' || IsFullA64Enabled()) then
        // Trapping of the exception enabled.
        // It is IMPLEMENTATION DEFINED whether the enable bit may be set at all,
        // and if so then how exceptions and in what order that they may be
        // accumulated before calling FPTrappedException().
        bits(8) accumulated_exceptions = GetAccumulatedFPExceptions();
        accumulated_exceptions<cumul> = '1';
        if boolean IMPLEMENTATION_DEFINED "Support trapping of floating-point exceptions" then
            if UsingAArch32() then
                AArch32.FPTrappedException(accumulated_exceptions);
            else
                is_ase = IsASEInstruction();
                AArch64.FPTrappedException(is_ase, accumulated_exceptions);
        else
            // The exceptions generated by this instruction are accumulated by the PE and
            // FPTrappedException is called later during its execution, before the next
            // instruction is executed. This field is cleared at the start of each FP instruction.
            SetAccumulatedFPExceptions(accumulated_exceptions);
    elsif UsingAArch32() then
        // Set the cumulative exception bit
        FPSCR<cumul> = '1';
    else
        // Set the cumulative exception bit
        FPSR<cumul> = '1';

    return;
```

Library pseudocode for shared/functions/float/fpprocessnan/FPProcessNaN

```
// FPProcessNaN()
// ==============

bits(N) FPProcessNaN(FPType fptype, bits(N) op, FPCRType fpcr)
    boolean fpexc = TRUE;   // Generate floating-point exceptions
    return FPProcessNaN(fptype, op, fpcr, fpexc);

// FPProcessNaN()
// ==============
// Handle NaN input operands, returning the operand or default NaN value
// if fpcr.DN is selected. The 'fpcr' argument supplies the FPCR control bits.
// The 'fpexc' argument controls the generation of exceptions, regardless of
// whether 'fptype' is a signalling NaN or a quiet NaN.

bits(N) FPProcessNaN(FPType fptype, bits(N) op, FPCRType fpcr, boolean fpexc)

    assert N IN {16,32,64};
    assert fptype IN {FPType_QNaN, FPType_SNaN};
    integer topfrac;

    case N of
        when 16 topfrac =  9;
        when 32 topfrac = 22;
        when 64 topfrac = 51;

    result = op;
    if fptype == FPType_SNaN then
        result<topfrac> = '1';
        if fpexc then FPProcessException(FPExc_InvalidOp, fpcr);
    if fpcr.DN == '1' then  // DefaultNaN requested
        result = FPDefaultNaN(fpcr, N);

    return result;
```

Library pseudocode for shared/functions/float/fpprocessnans/FPProcessNaNs

```
// FPProcessNaNs()
// ===============

(boolean, bits(N)) FPProcessNaNs(FPType type1, FPType type2, bits(N) op1,
                                 bits(N) op2, FPCRType fpcr)
    boolean fpexc       = TRUE;     // Generate floating-point exceptions
    return FPProcessNaNs(type1, type2, op1, op2, fpcr, fpexc);

// FPProcessNaNs()
// ===============
//
// The boolean part of the return value says whether a NaN has been found and
// processed. The bits(N) part is only relevant if it has and supplies the
// result of the operation.
//
// The 'fpcr' argument supplies FPCR control bits and 'altfmaxfmin' controls
// alternative floating-point behavior for FMAX, FMIN and variants. 'fpexc'
// controls the generation of floating-point exceptions. Status information
// is updated directly in the FPSR where appropriate.

(boolean, bits(N)) FPProcessNaNs(FPType type1, FPType type2, bits(N) op1, bits(N) op2,
                                 FPCRType fpcr, boolean fpexc)

    assert N IN {16,32,64};
    boolean done;
    bits(N) result;
    boolean altfp    = HaveAltFP() && !UsingAArch32() && fpcr.AH == '1';
    boolean op1_nan  = type1 IN {FPType_SNaN, FPType_QNaN};
    boolean op2_nan  = type2 IN {FPType_SNaN, FPType_QNaN};
    boolean any_snan = type1 == FPType_SNaN || type2 == FPType_SNaN;
    FPType  type_nan = if any_snan then FPType_SNaN else FPType_QNaN;

    if altfp && op1_nan && op2_nan then
        // <n> register NaN selected
        done = TRUE;  result = FPProcessNaN(type_nan, op1, fpcr, fpexc);
    elsif type1 == FPType_SNaN then
        done = TRUE;  result = FPProcessNaN(type1, op1, fpcr, fpexc);
    elsif type2 == FPType_SNaN then
        done = TRUE;  result = FPProcessNaN(type2, op2, fpcr, fpexc);
    elsif type1 == FPType_QNaN then
        done = TRUE;  result = FPProcessNaN(type1, op1, fpcr, fpexc);
    elsif type2 == FPType_QNaN then
        done = TRUE;  result = FPProcessNaN(type2, op2, fpcr, fpexc);
    else
        done = FALSE;  result = Zeros(N);  // 'Don't care' result

    return (done, result);
```

Library pseudocode for shared/functions/float/fpprocessnans3/FPProcessNaNs3

```
// FPProcessNaNs3()
// ================

(boolean, bits(N)) FPProcessNaNs3(FPType type1, FPType type2, FPType type3,
                                  bits(N) op1, bits(N) op2, bits(N) op3,
                                  FPCRType fpcr)
    boolean fpexc = TRUE;   // Generate floating-point exceptions
    return FPProcessNaNs3(type1, type2, type3, op1, op2, op3, fpcr, fpexc);

// FPProcessNaNs3()
// ================
// The boolean part of the return value says whether a NaN has been found and
// processed. The bits(N) part is only relevant if it has and supplies the
// result of the operation.
//
// The 'fpcr' argument supplies FPCR control bits and 'fpexc' controls the
// generation of floating-point exceptions. Status information is updated
// directly in the FPSR where appropriate.

(boolean, bits(N)) FPProcessNaNs3(FPType type1, FPType type2, FPType type3,
                                  bits(N) op1, bits(N) op2, bits(N) op3,
                                  FPCRType fpcr, boolean fpexc)

    assert N IN {16,32,64};
    bits(N) result;
    boolean op1_nan = type1 IN {FPType_SNaN, FPType_QNaN};
    boolean op2_nan = type2 IN {FPType_SNaN, FPType_QNaN};
    boolean op3_nan = type3 IN {FPType_SNaN, FPType_QNaN};

    boolean altfp = HaveAltFP() && !UsingAArch32() && fpcr.AH == '1';
    FPType type_nan;
    if altfp then
        if type1 == FPType_SNaN || type2 == FPType_SNaN || type3 == FPType_SNaN then
            type_nan = FPType_SNaN;
        else
            type_nan = FPType_QNaN;

    boolean done;
    if altfp && op1_nan && op2_nan && op3_nan then
        // <n> register NaN selected
        done = TRUE;  result = FPProcessNaN(type_nan, op2, fpcr, fpexc);
    elsif altfp && op2_nan && (op1_nan || op3_nan) then
        // <n> register NaN selected
        done = TRUE;  result = FPProcessNaN(type_nan, op2, fpcr, fpexc);
    elsif altfp && op3_nan && op1_nan then
        // <m> register NaN selected
        done = TRUE;  result = FPProcessNaN(type_nan, op3, fpcr, fpexc);
    elsif type1 == FPType_SNaN then
        done = TRUE;  result = FPProcessNaN(type1, op1, fpcr, fpexc);
    elsif type2 == FPType_SNaN then
        done = TRUE;  result = FPProcessNaN(type2, op2, fpcr, fpexc);
    elsif type3 == FPType_SNaN then
        done = TRUE;  result = FPProcessNaN(type3, op3, fpcr, fpexc);
    elsif type1 == FPType_QNaN then
        done = TRUE;  result = FPProcessNaN(type1, op1, fpcr, fpexc);
    elsif type2 == FPType_QNaN then
        done = TRUE;  result = FPProcessNaN(type2, op2, fpcr, fpexc);
    elsif type3 == FPType_QNaN then
        done = TRUE;  result = FPProcessNaN(type3, op3, fpcr, fpexc);
    else
        done = FALSE;  result = Zeros(N);  // 'Don't care' result

    return (done, result);
```

Library pseudocode for shared/functions/float/fpprocessnans4/FPProcessNaNs4

```
// FPProcessNaNs4()
// ================
// The boolean part of the return value says whether a NaN has been found and
// processed. The bits(N) part is only relevant if it has and supplies the
// result of the operation.
//
// The 'fpcr' argument supplies FPCR control bits.
// Status information is updated directly in the FPSR where appropriate.
// The 'fpexc' controls the generation of floating-point exceptions.

(boolean, bits(N)) FPProcessNaNs4(FPType type1, FPType type2, FPType type3, FPType type4,
                                  bits(N DIV 2) op1, bits(N DIV 2) op2, bits(N DIV 2) op3,
                                  bits(N DIV 2) op4, FPCRType fpcr, boolean fpexc)

    assert N == 32;

    bits(N) result;
    boolean done;
    // The FPCR.AH control does not affect these checks
    if type1 == FPType_SNaN then
        done = TRUE;  result = FPConvertNaN(FPProcessNaN(type1, op1, fpcr, fpexc), N);
    elsif type2 == FPType_SNaN then
        done = TRUE;  result = FPConvertNaN(FPProcessNaN(type2, op2, fpcr, fpexc), N);
    elsif type3 == FPType_SNaN then
        done = TRUE;  result = FPConvertNaN(FPProcessNaN(type3, op3, fpcr, fpexc), N);
    elsif type4 == FPType_SNaN then
        done = TRUE;  result = FPConvertNaN(FPProcessNaN(type4, op4, fpcr, fpexc), N);
    elsif type1 == FPType_QNaN then
        done = TRUE;  result = FPConvertNaN(FPProcessNaN(type1, op1, fpcr, fpexc), N);
    elsif type2 == FPType_QNaN then
        done = TRUE;  result = FPConvertNaN(FPProcessNaN(type2, op2, fpcr, fpexc), N);
    elsif type3 == FPType_QNaN then
        done = TRUE;  result = FPConvertNaN(FPProcessNaN(type3, op3, fpcr, fpexc), N);
    elsif type4 == FPType_QNaN then
        done = TRUE;  result = FPConvertNaN(FPProcessNaN(type4, op4, fpcr, fpexc), N);
    else
        done = FALSE;  result = Zeros(N);  // 'Don't care' result

    return (done, result);
```

Library pseudocode for shared/functions/float/fprecipestimate/FPRecipEstimate

```
// FPRecipEstimate()
// =================

bits(N) FPRecipEstimate(bits(N) operand, FPCRType fpcr_in)

    assert N IN {16,32,64};
    FPCRType fpcr = fpcr_in;
    bits(N) result;
    boolean overflow_to_inf;
    // When using alternative floating-point behavior, do not generate
    // floating-point exceptions, flush denormal input and output to zero,
    // and use RNE rounding mode.
    boolean altfp = HaveAltFP() && !UsingAArch32() && fpcr.AH == '1';
    boolean fpexc = !altfp;
    if altfp then fpcr.<FIZ,FZ> = '11';
    if altfp then fpcr.RMode    = '00';

    (fptype,sign,value) = FPUnpack(operand, fpcr, fpexc);

    FPRounding rounding = FPRoundingMode(fpcr);
    if fptype == FPType_SNaN || fptype == FPType_QNaN then
        result = FPProcessNaN(fptype, operand, fpcr, fpexc);
    elsif fptype == FPType_Infinity then
        result = FPZero(sign, N);
    elsif fptype == FPType_Zero then
        result = FPInfinity(sign, N);
        if fpexc then FPProcessException(FPExc_DivideByZero, fpcr);
    elsif (
            (N == 16 && Abs(value) < 2.0^-16) ||
            (N == 32 && Abs(value) < 2.0^-128) ||
            (N == 64 && Abs(value) < 2.0^-1024)
          ) then
        case rounding of
            when FPRounding_TIEEVEN
                overflow_to_inf = TRUE;
            when FPRounding_POSINF
                overflow_to_inf = (sign == '0');
            when FPRounding_NEGINF
                overflow_to_inf = (sign == '1');
            when FPRounding_ZERO
                overflow_to_inf = FALSE;
        result = if overflow_to_inf then FPInfinity(sign, N) else FPMaxNormal(sign, N);
        if fpexc then
            FPProcessException(FPExc_Overflow, fpcr);
            FPProcessException(FPExc_Inexact, fpcr);
    elsif ((fpcr.FZ == '1' && N != 16) || (fpcr.FZ16 == '1' && N == 16))
          && (
               (N == 16 && Abs(value) >= 2.0^14) ||
               (N == 32 && Abs(value) >= 2.0^126) ||
               (N == 64 && Abs(value) >= 2.0^1022)
             ) then
        // Result flushed to zero of correct sign
        result = FPZero(sign, N);

        // Flush-to-zero never generates a trapped exception.
        if UsingAArch32() then
            FPSCR.UFC = '1';
        else
            if fpexc then FPSR.UFC = '1';
    else
        // Scale to a fixed point value in the range 0.5 <= x < 1.0 in steps of 1/512, and
        // calculate result exponent. Scaled value has copied sign bit,
        // exponent = 1022 = double-precision biased version of -1,
        // fraction = original fraction
        bits(52) fraction;
        integer exp;
        case N of
            when 16
                fraction = operand<9:0> : Zeros(42);
                exp = UInt(operand<14:10>);
```

```
        when 32
            fraction = operand<22:0> : Zeros(29);
            exp = UInt(operand<30:23>);
        when 64
            fraction = operand<51:0>;
            exp = UInt(operand<62:52>);

    if exp == 0 then
        if fraction<51> == '0' then
            exp = -1;
            fraction = fraction<49:0>:'00';
        else
            fraction = fraction<50:0>:'0';

    integer scaled;
    boolean increasedprecision = N==32 && HaveFeatRPRES() && altfp;

    if !increasedprecision then
        scaled = UInt('1':fraction<51:44>);
    else
        scaled = UInt('1':fraction<51:41>);

    integer result_exp;
    case N of
        when 16 result_exp =   29 - exp; // In range 29-30 = -1 to 29+1 = 30
        when 32 result_exp =  253 - exp; // In range 253-254 = -1 to 253+1 = 254
        when 64 result_exp = 2045 - exp; // In range 2045-2046 = -1 to 2045+1 = 2046

    // Scaled is in range 256 .. 511 or 2048 .. 4095 range representing a
    // fixed-point number in range [0.5 .. 1.0].
    estimate = RecipEstimate(scaled, increasedprecision);

    // Estimate is in the range 256 .. 511 or 4096 .. 8191 representing a
    // fixed-point result in the range [1.0 .. 2.0].
    // Convert to scaled floating point result with copied sign bit,
    // high-order bits from estimate, and exponent calculated above.
    if !increasedprecision then
        fraction = estimate<7:0> : Zeros(44);
    else
        fraction = estimate<11:0> : Zeros(40);

    if result_exp == 0 then
        fraction = '1' : fraction<51:1>;
    elsif result_exp == -1 then
        fraction = '01' : fraction<51:2>;
        result_exp = 0;

    case N of
        when 16 result = sign : result_exp<N-12:0> : fraction<51:42>;
        when 32 result = sign : result_exp<N-25:0> : fraction<51:29>;
        when 64 result = sign : result_exp<N-54:0> : fraction<51:0>;

return result;
```

Library pseudocode for shared/functions/float/fprecipestimate/RecipEstimate

```
// RecipEstimate()
// ===============
// Compute estimate of reciprocal of 9-bit fixed-point number.
//
// a is in range 256 .. 511 or 2048 .. 4096 representing a number in
// the range 0.5 <= x < 1.0.
// increasedprecision determines if the mantissa is 8-bit or 12-bit.
// result is in the range 256 .. 511 or 4096 .. 8191 representing a
// number in the range 1.0 to 511/256 or 1.00 to 8191/4096.

integer RecipEstimate(integer a_in, boolean increasedprecision)

    integer a = a_in;
    integer r;
    if !increasedprecision then
        assert 256 <= a && a < 512;
        a = a*2+1;                       // Round to nearest
        integer b = (2 ^ 19) DIV a;
        r = (b+1) DIV 2;                 // Round to nearest
        assert 256 <= r && r < 512;
    else
        assert 2048 <= a && a < 4096;
        a = a*2+1;                       // Round to nearest
        real real_val = Real(2^25)/Real(a);
        r = RoundDown(real_val);
        real error = real_val - Real(r);
        boolean round_up = error > 0.5;  // Error cannot be exactly 0.5 so do not need tie case
        if round_up then r = r+1;
        assert 4096 <= r && r < 8192;

    return r;
```

Library pseudocode for shared/functions/float/fprecpx/FPRecpX

```
// FPRecpX()
// =========

bits(N) FPRecpX(bits(N) op, FPCRType fpcr_in)

    assert N IN {16,32,64};
    FPCRType fpcr = fpcr_in;
    integer esize;
    case N of
        when 16 esize =  5;
        when 32 esize =  8;
        when 64 esize = 11;

    bits(N)           result;
    bits(esize)       exp;
    bits(esize)       max_exp;
    bits(N-(esize+1)) frac = Zeros(N-(esize+1));

    boolean altfp = HaveAltFP() && fpcr.AH == '1';
    boolean fpexc = !altfp;                 // Generate no floating-point exceptions
    if altfp then fpcr.<FIZ,FZ> = '11';     // Flush denormal input and output to zero
    (fptype,sign,value) = FPUnpack(op, fpcr, fpexc);

    case N of
        when 16 exp = op<(10+esize)-1:10>;
        when 32 exp = op<(23+esize)-1:23>;
        when 64 exp = op<(52+esize)-1:52>;

    max_exp = Ones(esize) - 1;

    if fptype == FPType_SNaN || fptype == FPType_QNaN then
        result = FPProcessNaN(fptype, op, fpcr, fpexc);
    else
        if IsZero(exp) then                 // Zero and denormals
            result = sign:max_exp:frac;
        else                                // Infinities and normals
            result = sign:NOT(exp):frac;

    return result;
```

Library pseudocode for shared/functions/float/fpround/FPRound

```
// FPRound()
// =========
// Generic conversion from precise, unbounded real data type to IEEE format.

bits(N) FPRound(real op, FPCRType fpcr, integer N)
    return FPRound(op, fpcr, FPRoundingMode(fpcr), N);

// FPRound()
// =========
// For directed FP conversion, includes an explicit 'rounding' argument.

bits(N) FPRound(real op, FPCRType fpcr_in, FPRounding rounding, integer N)
    boolean fpexc = TRUE;    // Generate floating-point exceptions
    return FPRound(op, fpcr_in, rounding, fpexc, N);

// FPRound()
// =========
// For AltFP, includes an explicit FPEXC argument to disable exception
// generation and switches off Arm alternate half-precision mode.

bits(N) FPRound(real op, FPCRType fpcr_in, FPRounding rounding, boolean fpexc, integer N)
    FPCRType fpcr = fpcr_in;
    fpcr.AHP = '0';
    boolean isbfloat16 = FALSE;
    return FPRoundBase(op, fpcr, rounding, isbfloat16, fpexc, N);
```

Library pseudocode for shared/functions/float/fpround/FPRoundBase

```
// FPRoundBase()
// =============
// For BFloat16, includes an explicit 'isbfloat16' argument.

bits(N) FPRoundBase(real op, FPCRType fpcr, FPRounding rounding, boolean isbfloat16, integer N)
    boolean fpexc = TRUE;    // Generate floating-point exceptions
    return FPRoundBase(op, fpcr, rounding, isbfloat16, fpexc, N);

// FPRoundBase()
// =============
// Convert a real number 'op' into an N-bit floating-point value using the
// supplied rounding mode 'rounding'.
//
// The 'fpcr' argument supplies FPCR control bits and 'fpexc' controls the
// generation of floating-point exceptions. Status information is updated
// directly in the FPSR where appropriate.

bits(N) FPRoundBase(real op, FPCRType fpcr, FPRounding rounding,
                    boolean isbfloat16, boolean fpexc, integer N)

    assert N IN {16,32,64};
    assert op != 0.0;
    assert rounding != FPRounding_TIEAWAY;
    bits(N) result;

    // Obtain format parameters - minimum exponent, numbers of exponent and fraction bits.
    integer minimum_exp;
    integer F;
    integer E;
    if N == 16 then
        minimum_exp = -14;  E = 5;  F = 10;
    elsif N == 32 && isbfloat16 then
        minimum_exp = -126;  E = 8;  F = 7;
    elsif N == 32 then
        minimum_exp = -126;  E = 8;  F = 23;
    else  // N == 64
        minimum_exp = -1022;  E = 11;  F = 52;

    // Split value into sign, unrounded mantissa and exponent.
    bit sign;
    real mantissa;
    if op < 0.0 then
        sign = '1';  mantissa = -op;
    else
        sign = '0';  mantissa = op;
    exponent = 0;
    while mantissa < 1.0 do
        mantissa = mantissa * 2.0;  exponent = exponent - 1;
    while mantissa >= 2.0 do
        mantissa = mantissa / 2.0;  exponent = exponent + 1;

    // When TRUE, detection of underflow occurs after rounding and the test for a
    // denormalized number for single and double precision values occurs after rounding.
    altfp = HaveAltFP() && !UsingAArch32() && fpcr.AH == '1';

    // Deal with flush-to-zero before rounding if FPCR.AH != '1'.
    if (!altfp && ((fpcr.FZ == '1' && N != 16) || (fpcr.FZ16 == '1' && N == 16)) &&
        exponent < minimum_exp) then
        // Flush-to-zero never generates a trapped exception.
        if UsingAArch32() then
            FPSCR.UFC = '1';
        else
            if fpexc then FPSR.UFC = '1';
        return FPZero(sign, N);

    biased_exp_unconstrained = (exponent - minimum_exp) + 1;
    int_mant_unconstrained = RoundDown(mantissa * 2.0^F);
    error_unconstrained = mantissa * 2.0^F - Real(int_mant_unconstrained);

    // Start creating the exponent value for the result. Start by biasing the actual exponent
```

```
    // so that the minimum exponent becomes 1, lower values 0 (indicating possible underflow).
    biased_exp = Max((exponent - minimum_exp) + 1, 0);
    if biased_exp == 0 then mantissa = mantissa / 2.0^(minimum_exp - exponent);

    // Get the unrounded mantissa as an integer, and the "units in last place" rounding error.
    int_mant = RoundDown(mantissa * 2.0^F);  // < 2.0^F if biased_exp == 0, >= 2.0^F if not
    error = mantissa * 2.0^F - Real(int_mant);

    // Underflow occurs if exponent is too small before rounding, and result is inexact or
    // the Underflow exception is trapped. This applies before rounding if FPCR.AH != '1'.
    boolean trapped_UF = fpcr.UFE == '1' && (!InStreamingMode() || IsFullA64Enabled());
    if !altfp && biased_exp == 0 && (error != 0.0 || trapped_UF) then
        if fpexc then FPProcessException(FPExc_Underflow, fpcr);

    // Round result according to rounding mode.
    boolean round_up_unconstrained;
    boolean round_up;
    boolean overflow_to_inf;
    if altfp then

        case rounding of
            when FPRounding_TIEEVEN
                round_up_unconstrained = (error_unconstrained > 0.5 ||
                   (error_unconstrained == 0.5 && int_mant_unconstrained<0> == '1'));
                round_up = (error > 0.5 || (error == 0.5 && int_mant<0> == '1'));
                overflow_to_inf = TRUE;
            when FPRounding_POSINF
                round_up_unconstrained = (error_unconstrained != 0.0 && sign == '0');
                round_up = (error != 0.0 && sign == '0');
                overflow_to_inf = (sign == '0');
            when FPRounding_NEGINF
                round_up_unconstrained = (error_unconstrained != 0.0 && sign == '1');
                round_up = (error != 0.0 && sign == '1');
                overflow_to_inf = (sign == '1');
            when FPRounding_ZERO, FPRounding_ODD
                round_up_unconstrained = FALSE;
                round_up = FALSE;
                overflow_to_inf = FALSE;

        if round_up_unconstrained then
            int_mant_unconstrained = int_mant_unconstrained + 1;
            if int_mant_unconstrained == 2^(F+1) then    // Rounded up to next exponent
                biased_exp_unconstrained = biased_exp_unconstrained + 1;
                int_mant_unconstrained   = int_mant_unconstrained DIV 2;

        // Deal with flush-to-zero and underflow after rounding if FPCR.AH == '1'.
        if biased_exp_unconstrained < 1 && int_mant_unconstrained != 0 then
            // the result of unconstrained rounding is less than the minimum normalized number
            if (fpcr.FZ == '1' && N != 16) || (fpcr.FZ16 == '1' && N == 16) then   // Flush-to-zero
                if fpexc then
                    FPSR.UFC = '1';
                    FPProcessException(FPExc_Inexact, fpcr);
                return FPZero(sign, N);
            elsif error != 0.0 || trapped_UF then
                if fpexc then FPProcessException(FPExc_Underflow, fpcr);
    else    // altfp == FALSE
        case rounding of
            when FPRounding_TIEEVEN
                round_up = (error > 0.5 || (error == 0.5 && int_mant<0> == '1'));
                overflow_to_inf = TRUE;
            when FPRounding_POSINF
                round_up = (error != 0.0 && sign == '0');
                overflow_to_inf = (sign == '0');
            when FPRounding_NEGINF
                round_up = (error != 0.0 && sign == '1');
                overflow_to_inf = (sign == '1');
            when FPRounding_ZERO, FPRounding_ODD
                round_up = FALSE;
                overflow_to_inf = FALSE;
```

```
    if round_up then
        int_mant = int_mant + 1;
        if int_mant == 2^F then      // Rounded up from denormalized to normalized
            biased_exp = 1;
        if int_mant == 2^(F+1) then  // Rounded up to next exponent
            biased_exp = biased_exp + 1;
            int_mant = int_mant DIV 2;

    // Handle rounding to odd
    if error != 0.0 && rounding == FPRounding_ODD then
        int_mant<0> = '1';

    // Deal with overflow and generate result.
    if N != 16 || fpcr.AHP == '0' then  // Single, double or IEEE half precision
        if biased_exp >= 2^E - 1 then
            result = if overflow_to_inf then FPInfinity(sign, N) else FPMaxNormal(sign, N);
            if fpexc then FPProcessException(FPExc_Overflow, fpcr);
            error = 1.0;  // Ensure that an Inexact exception occurs
        else
            result = sign : biased_exp<E-1:0> : int_mant<F-1:0> : Zeros(N-(E+F+1));
    else                                     // Alternative half precision
        if biased_exp >= 2^E then
            result = sign : Ones(N-1);
            if fpexc then FPProcessException(FPExc_InvalidOp, fpcr);
            error = 0.0;  // Ensure that an Inexact exception does not occur
        else
            result = sign : biased_exp<E-1:0> : int_mant<F-1:0> : Zeros(N-(E+F+1));

    // Deal with Inexact exception.
    if error != 0.0 then
        if fpexc then FPProcessException(FPExc_Inexact, fpcr);

    return result;
```

Library pseudocode for shared/functions/float/fpround/FPRoundCV

```
// FPRoundCV()
// ===========
// Used for FP to FP conversion instructions.
// For half-precision data ignores FZ16 and observes AHP.

bits(N) FPRoundCV(real op, FPCRType fpcr_in, FPRounding rounding, integer N)
    FPCRType fpcr = fpcr_in;
    fpcr.FZ16 = '0';
    boolean fpexc = TRUE;    // Generate floating-point exceptions
    boolean isbfloat16 = FALSE;
    return FPRoundBase(op, fpcr, rounding, isbfloat16, fpexc, N);
```

Library pseudocode for shared/functions/float/fprounding/FPRounding

```
// FPRounding
// ==========
// The conversion and rounding functions take an explicit
// rounding mode enumeration instead of booleans or FPCR values.

enumeration FPRounding  {FPRounding_TIEEVEN, FPRounding_POSINF,
                         FPRounding_NEGINF,  FPRounding_ZERO,
                         FPRounding_TIEAWAY, FPRounding_ODD};
```

Library pseudocode for shared/functions/float/fproundingmode/FPRoundingMode

```
// FPRoundingMode()
// ================
// Return the current floating-point rounding mode.

FPRounding FPRoundingMode(FPCRType fpcr)
    return FPDecodeRounding(fpcr.RMode);
```

Library pseudocode for shared/functions/float/fproundint/FPRoundInt

```
// FPRoundInt()
// ============

// Round op to nearest integral floating point value using rounding mode in FPCR/FPSCR.
// If EXACT is TRUE, set FPSR.IXC if result is not numerically equal to op.

bits(N) FPRoundInt(bits(N) op, FPCRType fpcr, FPRounding rounding, boolean exact)

    assert rounding != FPRounding_ODD;
    assert N IN {16,32,64};

    // When alternative floating-point support is TRUE, do not generate
    // Input Denormal floating-point exceptions.
    altfp = HaveAltFP() && !UsingAArch32() && fpcr.AH == '1';
    fpexc = !altfp;

    // Unpack using FPCR to determine if subnormals are flushed-to-zero.
    (fptype,sign,value) = FPUnpack(op, fpcr, fpexc);

    bits(N) result;
    if fptype == FPType_SNaN || fptype == FPType_QNaN then
        result = FPProcessNaN(fptype, op, fpcr);
    elsif fptype == FPType_Infinity then
        result = FPInfinity(sign, N);
    elsif fptype == FPType_Zero then
        result = FPZero(sign, N);
    else
        // Extract integer component.
        int_result = RoundDown(value);
        error = value - Real(int_result);

        // Determine whether supplied rounding mode requires an increment.
        boolean round_up;
        case rounding of
            when FPRounding_TIEEVEN
                round_up = (error > 0.5 || (error == 0.5 && int_result<0> == '1'));
            when FPRounding_POSINF
                round_up = (error != 0.0);
            when FPRounding_NEGINF
                round_up = FALSE;
            when FPRounding_ZERO
                round_up = (error != 0.0 && int_result < 0);
            when FPRounding_TIEAWAY
                round_up = (error > 0.5 || (error == 0.5 && int_result >= 0));

        if round_up then int_result = int_result + 1;

        // Convert integer value into an equivalent real value.
        real_result = Real(int_result);

        // Re-encode as a floating-point value, result is always exact.
        if real_result == 0.0 then
            result = FPZero(sign, N);
        else
            result = FPRound(real_result, fpcr, FPRounding_ZERO, N);

        // Generate inexact exceptions.
        if error != 0.0 && exact then
            FPProcessException(FPExc_Inexact, fpcr);

    return result;
```

Library pseudocode for shared/functions/float/fproundintn/FPRoundIntN

```
// FPRoundIntN()
// =============

bits(N) FPRoundIntN(bits(N) op, FPCRType fpcr, FPRounding rounding, integer intsize)
    assert rounding != FPRounding_ODD;
    assert N IN {32,64};
    assert intsize IN {32, 64};
    integer exp;
    bits(N) result;
    boolean round_up;
    constant integer E = (if N == 32 then 8 else 11);
    constant integer F = N - (E + 1);

    // When alternative floating-point support is TRUE, do not generate
    // Input Denormal floating-point exceptions.
    altfp = HaveAltFP() && !UsingAArch32() && fpcr.AH == '1';
    fpexc = !altfp;

    // Unpack using FPCR to determine if subnormals are flushed-to-zero.
    (fptype,sign,value) = FPUnpack(op, fpcr, fpexc);

    if fptype IN {FPType_SNaN, FPType_QNaN, FPType_Infinity} then
        if N == 32 then
            exp = 126 + intsize;
            result = '1':exp<(E-1):0>:Zeros(F);
        else
            exp = 1022+intsize;
            result = '1':exp<(E-1):0>:Zeros(F);
        FPProcessException(FPExc_InvalidOp, fpcr);
    elsif fptype == FPType_Zero then
        result = FPZero(sign, N);
    else
        // Extract integer component.
        int_result = RoundDown(value);
        error = value - Real(int_result);

        // Determine whether supplied rounding mode requires an increment.
        case rounding of
            when FPRounding_TIEEVEN
                round_up = error > 0.5 || (error == 0.5 && int_result<0> == '1');
            when FPRounding_POSINF
                round_up = error != 0.0;
            when FPRounding_NEGINF
                round_up = FALSE;
            when FPRounding_ZERO
                round_up = error != 0.0 && int_result < 0;
            when FPRounding_TIEAWAY
                round_up = error > 0.5 || (error == 0.5 && int_result >= 0);

        if round_up then int_result = int_result + 1;
        overflow = int_result > 2^(intsize-1)-1 || int_result < -1*2^(intsize-1);

        if overflow then
            if N == 32 then
                exp = 126 + intsize;
                result = '1':exp<(E-1):0>:Zeros(F);
            else
                exp = 1022 + intsize;
                result = '1':exp<(E-1):0>:Zeros(F);
            FPProcessException(FPExc_InvalidOp, fpcr);
            // This case shouldn't set Inexact.
            error = 0.0;

        else
            // Convert integer value into an equivalent real value.
            real_result = Real(int_result);

            // Re-encode as a floating-point value, result is always exact.
            if real_result == 0.0 then
                result = FPZero(sign, N);
```

```
            else
                result = FPRound(real_result, fpcr, FPRounding_ZERO, N);

        // Generate inexact exceptions.
        if error != 0.0 then
            FPProcessException(FPExc_Inexact, fpcr);

    return result;
```

Library pseudocode for shared/functions/float/fprsqrtestimate/FPRSqrtEstimate

```
// FPRSqrtEstimate()
// =================

bits(N) FPRSqrtEstimate(bits(N) operand, FPCRType fpcr_in)

    assert N IN {16,32,64};
    FPCRType fpcr = fpcr_in;

    // When using alternative floating-point behavior, do not generate
    // floating-point exceptions and flush denormal input to zero.
    boolean altfp = HaveAltFP() && !UsingAArch32() && fpcr.AH == '1';
    boolean fpexc = !altfp;
    if altfp then fpcr.<FIZ,FZ> = '11';

    (fptype,sign,value) = FPUnpack(operand, fpcr, fpexc);

    bits(N) result;
    if fptype == FPType_SNaN || fptype == FPType_QNaN then
        result = FPProcessNaN(fptype, operand, fpcr, fpexc);
    elsif fptype == FPType_Zero then
        result = FPInfinity(sign, N);
        if fpexc then FPProcessException(FPExc_DivideByZero, fpcr);
    elsif sign == '1' then
        result = FPDefaultNaN(fpcr, N);
        if fpexc then FPProcessException(FPExc_InvalidOp, fpcr);
    elsif fptype == FPType_Infinity then
        result = FPZero('0', N);
    else
        // Scale to a fixed-point value in the range 0.25 <= x < 1.0 in steps of 512, with the
        // evenness or oddness of the exponent unchanged, and calculate result exponent.
        // Scaled value has copied sign bit, exponent = 1022 or 1021 = double-precision
        // biased version of -1 or -2, fraction = original fraction extended with zeros.

        bits(52) fraction;
        integer exp;
        case N of
            when 16
                fraction = operand<9:0> : Zeros(42);
                exp = UInt(operand<14:10>);
            when 32
                fraction = operand<22:0> : Zeros(29);
                exp = UInt(operand<30:23>);
            when 64
                fraction = operand<51:0>;
                exp = UInt(operand<62:52>);

        if exp == 0 then
            while fraction<51> == '0' do
                fraction = fraction<50:0> : '0';
                exp = exp - 1;
            fraction = fraction<50:0> : '0';

        integer scaled;
        boolean increasedprecision = N==32 && HaveFeatRPRES() && altfp;

        if !increasedprecision then
            if exp<0> == '0' then
                scaled = UInt('1':fraction<51:44>);
            else
                scaled = UInt('01':fraction<51:45>);
        else
            if exp<0> == '0' then
                scaled = UInt('1':fraction<51:41>);
            else
                scaled = UInt('01':fraction<51:42>);

        integer result_exp;
        case N of
            when 16 result_exp = (  44 - exp) DIV 2;
            when 32 result_exp = ( 380 - exp) DIV 2;
```

```
        when 64 result_exp = (3068 - exp) DIV 2;

    estimate = RecipSqrtEstimate(scaled, increasedprecision);

    // Estimate is in the range 256 .. 511 or 4096 .. 8191 representing a
    // fixed-point result in the range [1.0 .. 2.0].
    // Convert to scaled floating point result with copied sign bit and high-order
    // fraction bits, and exponent calculated above.
    case N of
        when 16 result = '0' : result_exp<N-12:0> : estimate<7:0>:Zeros(2);
        when 32
            if !increasedprecision then
                result = '0' : result_exp<N-25:0> : estimate<7:0>:Zeros(15);
            else
                result = '0' : result_exp<N-25:0> : estimate<11:0>:Zeros(11);
        when 64 result = '0' : result_exp<N-54:0> : estimate<7:0>:Zeros(44);

return result;
```

Library pseudocode for shared/functions/float/fprsqrtestimate/RecipSqrtEstimate

```
// RecipSqrtEstimate()
// ===================
// Compute estimate of reciprocal square root of 9-bit fixed-point number.
//
// a_in is in range 128 .. 511 or 1024 .. 4095, with increased precision,
// representing a number in the range 0.25 <= x < 1.0.
// increasedprecision determines if the mantissa is 8-bit or 12-bit.
// result is in the range 256 .. 511 or 4096 .. 8191, with increased precision,
// representing a number in the range 1.0 to 511/256 or 8191/4096.

integer RecipSqrtEstimate(integer a_in, boolean increasedprecision)

    integer a = a_in;
    integer r;
    if !increasedprecision then
        assert 128 <= a && a < 512;
        if a < 256 then                         // 0.25 .. 0.5
            a = a*2+1;                          // a in units of 1/512 rounded to nearest
        else                                    // 0.5 .. 1.0
            a = (a >> 1) << 1;                  // Discard bottom bit
            a = (a+1)*2;                        // a in units of 1/256 rounded to nearest
        integer b = 512;
        while a*(b+1)*(b+1) < 2^28 do
            b = b+1;
        // b = largest b such that b < 2^14 / sqrt(a)
        r = (b+1) DIV 2;                        // Round to nearest
        assert 256 <= r && r < 512;
    else
        assert 1024 <= a && a < 4096;
        real real_val;
        real error;
        integer int_val;

        if a < 2048 then                        // 0.25... 0.5
            a = a*2 + 1;                        // Take 10 bits of fraction and force a 1 at the bottom
            real_val = Real(a)/2.0;
        else                                    // 0.5..1.0
            a = (a >> 1) << 1;                  // Discard bottom bit
            a = a+1;                            // Take 10 bits of fraction and force a 1 at the bottom
            real_val = Real(a);

        real_val = Sqrt(real_val);              // This number will lie in the range of 32 to 64
                                                // Round to nearest even for a DP float number
        real_val = real_val * Real(2^47);       // The integer is the size of the whole DP mantissa
        int_val  = RoundDown(real_val);         // Calculate rounding value
        error    = real_val - Real(int_val);
        round_up = error > 0.5;                 // Error cannot be exactly 0.5 so do not need tie case
        if round_up then int_val = int_val+1;

        real_val = Real(2^65)/Real(int_val); // Lies in the range 4096 <= real_val < 8192
        int_val  = RoundDown(real_val);         // Round that (to nearest even) to give integer
        error    = real_val - Real(int_val);
        round_up = (error > 0.5 || (error == 0.5 && int_val<0> == '1'));
        if round_up then int_val = int_val+1;

        r = int_val;
        assert 4096 <= r && r < 8192;

    return r;
```

Library pseudocode for shared/functions/float/fpsqrt/FPSqrt

```
// FPSqrt()
// ========

bits(N) FPSqrt(bits(N) op, FPCRType fpcr)

    assert N IN {16,32,64};
    (fptype,sign,value) = FPUnpack(op, fpcr);

    bits(N) result;
    if fptype == FPType_SNaN || fptype == FPType_QNaN then
        result = FPProcessNaN(fptype, op, fpcr);
    elsif fptype == FPType_Zero then
        result = FPZero(sign, N);
    elsif fptype == FPType_Infinity && sign == '0' then
        result = FPInfinity(sign, N);
    elsif sign == '1' then
        result = FPDefaultNaN(fpcr, N);
        FPProcessException(FPExc_InvalidOp, fpcr);
    else
        result = FPRound(Sqrt(value), fpcr, N);
        FPProcessDenorm(fptype, N, fpcr);

    return result;
```

Library pseudocode for shared/functions/float/fpsub/FPSub

```
// FPSub()
// =======

bits(N) FPSub(bits(N) op1, bits(N) op2, FPCRType fpcr)
    boolean fpexc = TRUE;       // Generate floating-point exceptions
    return FPSub(op1, op2, fpcr, fpexc);

// FPSub()
// =======

bits(N) FPSub(bits(N) op1, bits(N) op2, FPCRType fpcr, boolean fpexc)

    assert N IN {16,32,64};
    rounding = FPRoundingMode(fpcr);

    (type1,sign1,value1) = FPUnpack(op1, fpcr, fpexc);
    (type2,sign2,value2) = FPUnpack(op2, fpcr, fpexc);

    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr, fpexc);
    if !done then
        inf1 = (type1 == FPType_Infinity);
        inf2 = (type2 == FPType_Infinity);
        zero1 = (type1 == FPType_Zero);
        zero2 = (type2 == FPType_Zero);

        if inf1 && inf2 && sign1 == sign2 then
            result = FPDefaultNaN(fpcr, N);
            if fpexc then FPProcessException(FPExc_InvalidOp, fpcr);
        elsif (inf1 && sign1 == '0') || (inf2 && sign2 == '1') then
            result = FPInfinity('0', N);
        elsif (inf1 && sign1 == '1') || (inf2 && sign2 == '0') then
            result = FPInfinity('1', N);
        elsif zero1 && zero2 && sign1 == NOT(sign2) then
            result = FPZero(sign1, N);
        else
            result_value = value1 - value2;
            if result_value == 0.0 then  // Sign of exact zero result depends on rounding mode
                result_sign = if rounding == FPRounding_NEGINF then '1' else '0';
                result = FPZero(result_sign, N);
            else
                result = FPRound(result_value, fpcr, rounding, fpexc, N);

        if fpexc then FPProcessDenorms(type1, type2, N, fpcr);

    return result;
```

Library pseudocode for shared/functions/float/fpsub/FPSub_ZA

```
// FPSub_ZA()
// ==========
// Calculates op1-op2 for SME2 ZA-targeting instructions.

bits(N) FPSub_ZA(bits(N) op1, bits(N) op2, FPCRType fpcr_in)
    FPCRType fpcr = fpcr_in;
    boolean fpexc = FALSE;      // Do not generate floating-point exceptions
    fpcr.DN = '1';              // Generate default NaN values
    return FPSub(op1, op2, fpcr, fpexc);
```

Library pseudocode for shared/functions/float/fpthree/FPThree

```
// FPThree()
// =========

bits(N) FPThree(bit sign, integer N)

    assert N IN {16,32,64};
    constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
    constant integer F = N - (E + 1);
    exp  = '1':Zeros(E-1);
    frac = '1':Zeros(F-1);
    result = sign : exp : frac;

    return result;
```

Library pseudocode for shared/functions/float/fptofixed/FPToFixed

```
// FPToFixed()
// ===========

// Convert N-bit precision floating point 'op' to M-bit fixed point with
// FBITS fractional bits, controlled by UNSIGNED and ROUNDING.

bits(M) FPToFixed(bits(N) op, integer fbits, boolean unsigned, FPCRType fpcr,
                  FPRounding rounding, integer M)

    assert N IN {16,32,64};
    assert M IN {16,32,64};
    assert fbits >= 0;
    assert rounding != FPRounding_ODD;

    // When alternative floating-point support is TRUE, do not generate
    // Input Denormal floating-point exceptions.
    altfp = HaveAltFP() && !UsingAArch32() && fpcr.AH == '1';
    fpexc = !altfp;

    // Unpack using fpcr to determine if subnormals are flushed-to-zero.
    (fptype,sign,value) = FPUnpack(op, fpcr, fpexc);

    // If NaN, set cumulative flag or take exception.
    if fptype == FPType_SNaN || fptype == FPType_QNaN then
        FPProcessException(FPExc_InvalidOp, fpcr);

    // Scale by fractional bits and produce integer rounded towards minus-infinity.
    value = value * 2.0^fbits;
    int_result = RoundDown(value);
    error = value - Real(int_result);

    // Determine whether supplied rounding mode requires an increment.
    boolean round_up;
    case rounding of
        when FPRounding_TIEEVEN
            round_up = (error > 0.5 || (error == 0.5 && int_result<0> == '1'));
        when FPRounding_POSINF
            round_up = (error != 0.0);
        when FPRounding_NEGINF
            round_up = FALSE;
        when FPRounding_ZERO
            round_up = (error != 0.0 && int_result < 0);
        when FPRounding_TIEAWAY
            round_up = (error > 0.5 || (error == 0.5 && int_result >= 0));

    if round_up then int_result = int_result + 1;

    // Generate saturated result and exceptions.
    (result, overflow) = SatQ(int_result, M, unsigned);
    if overflow then
        FPProcessException(FPExc_InvalidOp, fpcr);
    elsif error != 0.0 then
        FPProcessException(FPExc_Inexact, fpcr);

    return result;
```

Library pseudocode for shared/functions/float/fptofixedjs/FPToFixedJS

```
// FPToFixedJS()
// =============

// Converts a double precision floating point input value
// to a signed integer, with rounding to zero.

(bits(N), bit) FPToFixedJS(bits(M) op, FPCRType fpcr, boolean Is64, integer N)

    assert M == 64 && N == 32;

    // If FALSE, never generate Input Denormal floating-point exceptions.
    fpexc_idenorm = !(HaveAltFP() && !UsingAArch32() && fpcr.AH == '1');

    // Unpack using fpcr to determine if subnormals are flushed-to-zero.
    (fptype,sign,value) = FPUnpack(op, fpcr, fpexc_idenorm);

    z = '1';
    // If NaN, set cumulative flag or take exception.
    if fptype == FPType_SNaN || fptype == FPType_QNaN then
        FPProcessException(FPExc_InvalidOp, fpcr);
        z = '0';

    int_result = RoundDown(value);
    error = value - Real(int_result);

    // Determine whether supplied rounding mode requires an increment.

    round_it_up = (error != 0.0 && int_result < 0);
    if round_it_up then int_result = int_result + 1;

    integer result;
    if int_result < 0 then
        result = int_result - 2^32*RoundUp(Real(int_result)/Real(2^32));
    else
        result = int_result - 2^32*RoundDown(Real(int_result)/Real(2^32));

    // Generate exceptions.
    if int_result < -(2^31) || int_result > (2^31)-1 then
        FPProcessException(FPExc_InvalidOp, fpcr);
        z = '0';
    elsif error != 0.0 then
        FPProcessException(FPExc_Inexact, fpcr);
        z = '0';
    elsif sign == '1' && value == 0.0 then
        z = '0';
    elsif sign == '0' && value == 0.0 && !IsZero(op<51:0>) then
        z = '0';

    if fptype == FPType_Infinity then result = 0;

    return (result<N-1:0>, z);
```

Library pseudocode for shared/functions/float/fptwo/FPTwo

```
// FPTwo()
// =======

bits(N) FPTwo(bit sign, integer N)

    assert N IN {16,32,64};
    constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
    constant integer F = N - (E + 1);
    exp  = '1':Zeros(E-1);
    frac = Zeros(F);
    result = sign : exp : frac;

    return result;
```

Library pseudocode for shared/functions/float/fptype/FPType

```
// FPType
// ======

enumeration FPType {FPType_Zero,
                    FPType_Denormal,
                    FPType_Nonzero,
                    FPType_Infinity,
                    FPType_QNaN,
                    FPType_SNaN};
```

Library pseudocode for shared/functions/float/fpunpack/FPUnpack

```
// FPUnpack()
// ==========

(FPType, bit, real) FPUnpack(bits(N) fpval, FPCRType fpcr_in)
    FPCRType fpcr = fpcr_in;
    fpcr.AHP = '0';
    boolean fpexc = TRUE;   // Generate floating-point exceptions
    (fp_type, sign, value) = FPUnpackBase(fpval, fpcr, fpexc);
    return (fp_type, sign, value);

// FPUnpack()
// ==========
//
// Used by data processing, int/fixed to FP and FP to int/fixed conversion instructions.
// For half-precision data it ignores AHP, and observes FZ16.

(FPType, bit, real) FPUnpack(bits(N) fpval, FPCRType fpcr_in, boolean fpexc)
    FPCRType fpcr = fpcr_in;
    fpcr.AHP = '0';
    (fp_type, sign, value) = FPUnpackBase(fpval, fpcr, fpexc);
    return (fp_type, sign, value);
```

Library pseudocode for shared/functions/float/fpunpack/FPUnpackBase

```
// FPUnpackBase()
// ==============

(FPType, bit, real) FPUnpackBase(bits(N) fpval, FPCRType fpcr, boolean fpexc)
    boolean isbfloat16 = FALSE;
    (fp_type, sign, value) = FPUnpackBase(fpval, fpcr, fpexc, isbfloat16);
    return (fp_type, sign, value);

// FPUnpackBase()
// ==============
//
// Unpack a floating-point number into its type, sign bit and the real number
// that it represents. The real number result has the correct sign for numbers
// and infinities, is very large in magnitude for infinities, and is 0.0 for
// NaNs. (These values are chosen to simplify the description of comparisons
// and conversions.)
//
// The 'fpcr_in' argument supplies FPCR control bits, 'fpexc' controls the
// generation of floating-point exceptions and 'isbfloat16' determines whether
// N=16 signifies BFloat16 or half-precision type. Status information is updated
// directly in the FPSR where appropriate.

(FPType, bit, real) FPUnpackBase(bits(N) fpval, FPCRType fpcr_in, boolean fpexc,
                                 boolean isbfloat16)

    assert N IN {16,32,64};

    FPCRType fpcr = fpcr_in;

    boolean altfp = HaveAltFP() && !UsingAArch32();
    boolean fiz   = altfp && fpcr.FIZ == '1';
    boolean fz    = fpcr.FZ == '1' && !(altfp && fpcr.AH == '1');
    real value;
    bit sign;
    FPType fptype;

    if N == 16 && !isbfloat16 then
        sign   = fpval<15>;
        exp16  = fpval<14:10>;
        frac16 = fpval<9:0>;
        if IsZero(exp16) then
            if IsZero(frac16) || fpcr.FZ16 == '1' then
                fptype = FPType_Zero;  value = 0.0;
            else
                fptype = FPType_Denormal;  value = 2.0^-14 * (Real(UInt(frac16)) * 2.0^-10);
        elsif IsOnes(exp16) && fpcr.AHP == '0' then  // Infinity or NaN in IEEE format
            if IsZero(frac16) then
                fptype = FPType_Infinity;  value = 2.0^1000000;
            else
                fptype = if frac16<9> == '1' then FPType_QNaN else FPType_SNaN;
                value = 0.0;
        else
            fptype = FPType_Nonzero;
            value = 2.0^(UInt(exp16)-15) * (1.0 + Real(UInt(frac16)) * 2.0^-10);

    elsif N == 32 || isbfloat16 then
        bits(8) exp32;
        bits(23) frac32;
        if isbfloat16 then
            sign   = fpval<15>;
            exp32  = fpval<14:7>;
            frac32 = fpval<6:0> : Zeros(16);
        else
            sign   = fpval<31>;
            exp32  = fpval<30:23>;
            frac32 = fpval<22:0>;

        if IsZero(exp32) then
            if IsZero(frac32) then
                // Produce zero if value is zero.
```

```
                fptype = FPType_Zero;  value = 0.0;
            elsif fz || fiz then        // Flush-to-zero if FIZ==1 or AH,FZ==01
                fptype = FPType_Zero;  value = 0.0;
                // Check whether to raise Input Denormal floating-point exception.
                // fpcr.FIZ==1 does not raise Input Denormal exception.
                if fz then
                    // Denormalized input flushed to zero
                    if fpexc then FPProcessException(FPExc_InputDenorm, fpcr);
            else
                fptype = FPType_Denormal;  value = 2.0^-126 * (Real(UInt(frac32)) * 2.0^-23);
        elsif IsOnes(exp32) then
            if IsZero(frac32) then
                fptype = FPType_Infinity;  value = 2.0^1000000;
            else
                fptype = if frac32<22> == '1' then FPType_QNaN else FPType_SNaN;
                value = 0.0;
        else
            fptype = FPType_Nonzero;
            value = 2.0^(UInt(exp32)-127) * (1.0 + Real(UInt(frac32)) * 2.0^-23);

    else // N == 64
        sign   = fpval<63>;
        exp64  = fpval<62:52>;
        frac64 = fpval<51:0>;

        if IsZero(exp64) then
            if IsZero(frac64) then
                // Produce zero if value is zero.
                fptype = FPType_Zero;  value = 0.0;
            elsif fz || fiz then        // Flush-to-zero if FIZ==1 or AH,FZ==01
                fptype = FPType_Zero;  value = 0.0;
                // Check whether to raise Input Denormal floating-point exception.
                // fpcr.FIZ==1 does not raise Input Denormal exception.
                if fz then
                    // Denormalized input flushed to zero
                    if fpexc then FPProcessException(FPExc_InputDenorm, fpcr);
            else
                fptype = FPType_Denormal;  value = 2.0^-1022 * (Real(UInt(frac64)) * 2.0^-52);
        elsif IsOnes(exp64) then
            if IsZero(frac64) then
                fptype = FPType_Infinity;  value = 2.0^1000000;
            else
                fptype = if frac64<51> == '1' then FPType_QNaN else FPType_SNaN;
                value = 0.0;
        else
            fptype = FPType_Nonzero;
            value = 2.0^(UInt(exp64)-1023) * (1.0 + Real(UInt(frac64)) * 2.0^-52);

    if sign == '1' then value = -value;

    return (fptype, sign, value);
```

Library pseudocode for shared/functions/float/fpunpack/FPUnpackCV

```
// FPUnpackCV()
// ============
//
// Used for FP to FP conversion instructions.
// For half-precision data ignores FZ16 and observes AHP.

(FPType, bit, real) FPUnpackCV(bits(N) fpval, FPCRType fpcr_in)
    FPCRType fpcr = fpcr_in;
    fpcr.FZ16 = '0';
    boolean fpexc = TRUE;   // Generate floating-point exceptions
    (fp_type, sign, value) = FPUnpackBase(fpval, fpcr, fpexc);
    return (fp_type, sign, value);
```

Library pseudocode for shared/functions/float/fpzero/FPZero

```
// FPZero()
// ========

bits(N) FPZero(bit sign, integer N)

    assert N IN {16,32,64};
    constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
    constant integer F = N - (E + 1);
    exp  = Zeros(E);
    frac = Zeros(F);
    result = sign : exp : frac;

    return result;
```

Library pseudocode for shared/functions/float/vfpexpandimm/VFPExpandImm

```
// VFPExpandImm()
// ==============

bits(N) VFPExpandImm(bits(8) imm8, integer N)

    assert N IN {16,32,64};
    constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
    constant integer F = (N - E) - 1;
    sign = imm8<7>;
    exp  = NOT(imm8<6>):Replicate(imm8<6>,E-3):imm8<5:4>;
    frac = imm8<3:0>:Zeros(F-4);
    result = sign : exp : frac;

    return result;
```

Library pseudocode for shared/functions/integer/AddWithCarry

```
// AddWithCarry()
// ==============
// Integer addition with carry input, returning result and NZCV flags

(bits(N), bits(4)) AddWithCarry(bits(N) x, bits(N) y, bit carry_in)
    integer unsigned_sum = UInt(x) + UInt(y) + UInt(carry_in);
    integer signed_sum = SInt(x) + SInt(y) + UInt(carry_in);
    bits(N) result = unsigned_sum<N-1:0>; // same value as signed_sum<N-1:0>
    bit n = result<N-1>;
    bit z = if IsZero(result) then '1' else '0';
    bit c = if UInt(result) == unsigned_sum then '0' else '1';
    bit v = if SInt(result) == signed_sum then '0' else '1';
    return (result, n:z:c:v);
```

Library pseudocode for shared/functions/interrupts/InterruptID

```
// InterruptID
// ===========

enumeration InterruptID {
    InterruptID_PMUIRQ,
    InterruptID_COMMIRQ,
    InterruptID_CTIIRQ,
    InterruptID_COMMRX,
    InterruptID_COMMTX,
    InterruptID_CNTP,
    InterruptID_CNTHP,
    InterruptID_CNTHPS,
    InterruptID_CNTPS,
    InterruptID_CNTV,
    InterruptID_CNTHV,
    InterruptID_CNTHVS,
    InterruptID_PMBIRQ,
};
```

Library pseudocode for shared/functions/interrupts/SetInterruptRequestLevel

```
// SetInterruptRequestLevel()
// ==========================
// Set a level-sensitive interrupt to the specified level.

SetInterruptRequestLevel(InterruptID id, Signal level);
```

Library pseudocode for shared/functions/memory/AArch64.BranchAddr

```
// AArch64.BranchAddr()
// ====================
// Return the virtual address with tag bits removed.
// This is typically used when the address will be stored to the program counter.

bits(64) AArch64.BranchAddr(bits(64) vaddress, bits(2) el)
    assert !UsingAArch32();
    msbit = AddrTop(vaddress, TRUE, el);
    if msbit == 63 then
        return vaddress;
    elsif (el IN {EL0, EL1} || IsInHost()) && vaddress<msbit> == '1' then
        return SignExtend(vaddress<msbit:0>, 64);
    else
        return ZeroExtend(vaddress<msbit:0>, 64);
```

Library pseudocode for shared/functions/memory/AccessDescriptor

```
// AccessDescriptor
// ================
// Memory access or translation invocation details that steer architectural behavior

type AccessDescriptor is (
    AccessType acctype,
    bits(2) el,              // Acting EL for the access
    SecurityState ss,        // Acting Security State for the access
    boolean acqsc,           // Acquire with Sequential Consistency
    boolean acqpc,           // FEAT_LRCPC: Acquire with Processor Consistency
    boolean relsc,           // Release with Sequential Consistency
    boolean limitedordered, // FEAT_LOR: Acquire/Release with limited ordering
    boolean exclusive,       // Access has Exclusive semantics
    boolean atomicop,        // FEAT_LSE: Atomic read-modify-write access
    MemAtomicOp modop,       // FEAT_LSE: The modification operation in the 'atomicop' access
    boolean nontemporal,     // Hints the access is non-temporal
    boolean read,            // Read from memory or only require read permissions
    boolean write,           // Write to memory or only require write permissions
    CacheOp cacheop,         // DC/IC: Cache operation
    CacheOpScope opscope,    // DC/IC: Scope of cache operation
    CacheType cachetype,     // DC/IC: Type of target cache
    boolean pan,             // FEAT_PAN: The access is subject to PSTATE.PAN
    boolean transactional,  // FEAT_TME: Access is part of a transaction
    boolean nonfault,        // SVE: Non-faulting load
    boolean firstfault,      // SVE: First-fault load
    boolean first,           // SVE: First-fault load for the first active element
    boolean contiguous,      // SVE: Contiguous load/store not gather load/scatter store
    boolean streamingsve,    // SME: Access made by PE while in streaming SVE mode
    boolean ls64,            // FEAT_LS64: Accesses by accelerator support loads/stores
    boolean mops,            // FEAT_MOPS: Memory operation (CPY/SET) accesses
    boolean rcw,             // FEAT_THE: Read-Check-Write access
    boolean rcws,            // FEAT_THE: Read-Check-Write Software access
    boolean toplevel,        // FEAT_THE: Translation table walk access for TTB address
    VARange varange,         // FEAT_THE: The corresponding TTBR supplying the TTB
    boolean a32lsmd,         // A32 Load/Store Multiple Data access
    boolean tagchecked,      // FEAT_MTE2: Access is tag checked
    boolean tagaccess,       // FEAT_MTE: Access targets the tag bits
    MPAMinfo mpam            // FEAT_MPAM: MPAM information
)
```

Library pseudocode for shared/functions/memory/AccessType

```
// AccessType
// ==========

enumeration AccessType {
    AccessType_IFETCH,  // Instruction FETCH
    AccessType_GPR,     // Software load/store to a General Purpose Register
    AccessType_ASIMD,   // Software ASIMD extension load/store instructions
    AccessType_SVE,     // Software SVE load/store instructions
    AccessType_SME,     // Software SME load/store instructions
    AccessType_IC,      // Sysop IC
    AccessType_DC,      // Sysop DC (not DC {Z,G,GZ}VA)
    AccessType_DCZero,  // Sysop DC {Z,G,GZ}VA
    AccessType_AT,      // Sysop AT
    AccessType_NV2,     // NV2 memory redirected access
    AccessType_SPE,     // Statistical Profiling buffer access
    AccessType_GCS,     // Guarded Control Stack access
    AccessType_TRBE,    // Trace Buffer access
    AccessType_GPTW,    // Granule Protection Table Walk
    AccessType_TTW      // Translation Table Walk
};
```

Library pseudocode for shared/functions/memory/AddrTop

```
// AddrTop()
// =========
// Return the MSB number of a virtual address in the stage 1 translation regime for "el".
// If EL1 is using AArch64 then addresses from EL0 using AArch32 are zero-extended to 64 bits.

integer AddrTop(bits(64) address, boolean IsInstr, bits(2) el)
    assert HaveEL(el);
    regime = S1TranslationRegime(el);
    if ELUsingAArch32(regime) then
        // AArch32 translation regime.
        return 31;
    else
        if EffectiveTBI(address, IsInstr, el) == '1' then
            return 55;
        else
            return 63;
```

Library pseudocode for shared/functions/memory/AlignmentEnforced

```
// AlignmentEnforced()
// ===================
// For the active translation regime, determine if alignment is required by all accesses

boolean AlignmentEnforced()
    Regime regime = TranslationRegime(PSTATE.EL);

    bit A;
    case regime of
        when Regime_EL3  A = SCTLR_EL3.A;
        when Regime_EL30 A = SCTLR.A;
        when Regime_EL2  A = if ELUsingAArch32(EL2) then HSCTLR.A else SCTLR_EL2.A;
        when Regime_EL20 A = SCTLR_EL2.A;
        when Regime_EL10 A = if ELUsingAArch32(EL1) then SCTLR.A  else SCTLR_EL1.A;
        otherwise Unreachable();

    return A == '1';
```

Library pseudocode for shared/functions/memory/Allocation

```
constant bits(2) MemHint_No = '00';     // No Read-Allocate, No Write-Allocate
constant bits(2) MemHint_WA = '01';     // No Read-Allocate, Write-Allocate
constant bits(2) MemHint_RA = '10';     // Read-Allocate, No Write-Allocate
constant bits(2) MemHint_RWA = '11';    // Read-Allocate, Write-Allocate
```

Library pseudocode for shared/functions/memory/BigEndian

```
// BigEndian()
// ===========

boolean BigEndian(AccessType acctype)
    boolean bigend;
    if HaveNV2Ext() && acctype == AccessType_NV2 then
        return SCTLR_EL2.EE == '1';

    if UsingAArch32() then
        bigend = (PSTATE.E != '0');
    elsif PSTATE.EL == EL0 then
        bigend = (SCTLR[].E0E != '0');
    else
        bigend = (SCTLR[].EE != '0');
    return bigend;
```

Library pseudocode for shared/functions/memory/BigEndianReverse

```
// BigEndianReverse()
// ==================

bits(width) BigEndianReverse (bits(width) value)
    assert width IN {8, 16, 32, 64, 128};
    integer half = width DIV 2;
    if width == 8 then return value;
    return BigEndianReverse(value<half-1:0>) : BigEndianReverse(value<width-1:half>);
```

Library pseudocode for shared/functions/memory/Cacheability

```
constant bits(2) MemAttr_NC = '00';     // Non-cacheable
constant bits(2) MemAttr_WT = '10';     // Write-through
constant bits(2) MemAttr_WB = '11';     // Write-back
```

Library pseudocode for shared/functions/memory/CreateAccDescA32LSMD

```
// CreateAccDescA32LSMD()
// ======================
// Access descriptor for A32 loads/store multiple general purpose registers

AccessDescriptor CreateAccDescA32LSMD(MemOp memop)
    AccessDescriptor accdesc = NewAccDesc(AccessType_GPR);

    accdesc.read            = memop == MemOp_LOAD;
    accdesc.write           = memop == MemOp_STORE;
    accdesc.pan             = TRUE;
    accdesc.a32lsmd         = TRUE;
    accdesc.transactional   = HaveTME() && TSTATE.depth > 0;

    return accdesc;
```

Library pseudocode for shared/functions/memory/CreateAccDescASIMD

```
// CreateAccDescASIMD()
// ====================
// Access descriptor for ASIMD&FP loads/stores

AccessDescriptor CreateAccDescASIMD(MemOp memop, boolean nontemporal, boolean tagchecked)
    AccessDescriptor accdesc = NewAccDesc(AccessType_ASIMD);

    accdesc.nontemporal     = nontemporal;
    accdesc.read            = memop == MemOp_LOAD;
    accdesc.write           = memop == MemOp_STORE;
    accdesc.pan             = TRUE;
    accdesc.streamingsve    = InStreamingMode();
    if (accdesc.streamingsve && boolean IMPLEMENTATION_DEFINED
          "No tag checking of SIMD&FP loads and stores in Streaming SVE mode") then
        accdesc.tagchecked  = FALSE;
    else
        accdesc.tagchecked  = tagchecked;
    accdesc.transactional   = HaveTME() && TSTATE.depth > 0;

    return accdesc;
```

Library pseudocode for shared/functions/memory/CreateAccDescASIMDAcqRel

```
// CreateAccDescASIMDAcqRel()
// ==========================
// Access descriptor for ASIMD&FP loads/stores with ordering semantics

AccessDescriptor CreateAccDescASIMDAcqRel(MemOp memop, boolean tagchecked)
    AccessDescriptor accdesc = NewAccDesc(AccessType_ASIMD);

    accdesc.acqpc           = memop == MemOp_LOAD;
    accdesc.relsc           = memop == MemOp_STORE;
    accdesc.read            = memop == MemOp_LOAD;
    accdesc.write           = memop == MemOp_STORE;
    accdesc.pan             = TRUE;
    accdesc.streamingsve    = InStreamingMode();
    if (accdesc.streamingsve && boolean IMPLEMENTATION_DEFINED
          "No tag checking of SIMD&FP loads and stores in Streaming SVE mode") then
        accdesc.tagchecked  = FALSE;
    else
        accdesc.tagchecked  = tagchecked;
    accdesc.transactional   = HaveTME() && TSTATE.depth > 0;

    return accdesc;
```

Library pseudocode for shared/functions/memory/CreateAccDescAT

```
// CreateAccDescAT()
// =================
// Access descriptor for address translation operations

AccessDescriptor CreateAccDescAT(SecurityState ss, bits(2) el, boolean write, boolean pan)
    AccessDescriptor accdesc = NewAccDesc(AccessType_AT);

    accdesc.el              = el;
    accdesc.ss              = ss;
    accdesc.read            = !write;
    accdesc.write           = write;
    accdesc.pan             = pan;

    return accdesc;
```

Library pseudocode for shared/functions/memory/CreateAccDescAcqRel

```
// CreateAccDescAcqRel()
// =====================
// Access descriptor for general purpose register loads/stores with ordering semantics

AccessDescriptor CreateAccDescAcqRel(MemOp memop, boolean tagchecked)
    AccessDescriptor accdesc = NewAccDesc(AccessType_GPR);

    accdesc.acqsc           = memop == MemOp_LOAD;
    accdesc.relsc           = memop == MemOp_STORE;
    accdesc.read            = memop == MemOp_LOAD;
    accdesc.write           = memop == MemOp_STORE;
    accdesc.pan             = TRUE;
    accdesc.tagchecked      = tagchecked;
    accdesc.transactional   = HaveTME() && TSTATE.depth > 0;

    return accdesc;
```

Library pseudocode for shared/functions/memory/CreateAccDescAtomicOp

```
// CreateAccDescAtomicOp()
// =======================
// Access descriptor for atomic read-modify-write memory accesses

AccessDescriptor CreateAccDescAtomicOp(MemAtomicOp modop, boolean acquire, boolean release,
                                       boolean tagchecked)
    AccessDescriptor accdesc = NewAccDesc(AccessType_GPR);

    accdesc.acqsc           = acquire;
    accdesc.relsc           = release;
    accdesc.atomicop        = TRUE;
    accdesc.modop           = modop;
    accdesc.read            = TRUE;
    accdesc.write           = TRUE;
    accdesc.pan             = TRUE;
    accdesc.tagchecked      = tagchecked;
    accdesc.transactional   = HaveTME() && TSTATE.depth > 0;

    return accdesc;
```

Library pseudocode for shared/functions/memory/CreateAccDescDC

```
// CreateAccDescDC()
// =================
// Access descriptor for data cache operations

AccessDescriptor CreateAccDescDC(CacheRecord cache)
    AccessDescriptor accdesc = NewAccDesc(AccessType_DC);

    accdesc.cacheop         = cache.cacheop;
    accdesc.cachetype       = cache.cachetype;
    accdesc.opscope         = cache.opscope;

    return accdesc;
```

Library pseudocode for shared/functions/memory/CreateAccDescDCZero

```
// CreateAccDescDCZero()
// =====================
// Access descriptor for data cache zero operations

AccessDescriptor CreateAccDescDCZero(boolean tagaccess, boolean tagchecked)
    AccessDescriptor accdesc = NewAccDesc(AccessType_DCZero);

    accdesc.write           = TRUE;
    accdesc.pan             = TRUE;
    accdesc.tagchecked      = tagchecked;
    accdesc.tagaccess       = tagaccess;
    accdesc.transactional   = HaveTME() && TSTATE.depth > 0;

    return accdesc;
```

Library pseudocode for shared/functions/memory/CreateAccDescExLDST

```
// CreateAccDescExLDST()
// =====================
// Access descriptor for general purpose register loads/stores with exclusive semantics

AccessDescriptor CreateAccDescExLDST(MemOp memop, boolean acqrel, boolean tagchecked)
    AccessDescriptor accdesc = NewAccDesc(AccessType_GPR);

    accdesc.acqsc           = acqrel && memop == MemOp_LOAD;
    accdesc.relsc           = acqrel && memop == MemOp_STORE;
    accdesc.exclusive       = TRUE;
    accdesc.read            = memop == MemOp_LOAD;
    accdesc.write           = memop == MemOp_STORE;
    accdesc.pan             = TRUE;
    accdesc.tagchecked      = tagchecked;
    accdesc.transactional   = HaveTME() && TSTATE.depth > 0;

    return accdesc;
```

Library pseudocode for shared/functions/memory/CreateAccDescGCS

```
// CreateAccDescGCS()
// ==================
// Access descriptor for memory accesses to the Guarded Control Stack

AccessDescriptor CreateAccDescGCS(bits(2) el, MemOp memop)
    AccessDescriptor accdesc = NewAccDesc(AccessType_GCS);

    accdesc.el              = el;
    accdesc.read            = memop == MemOp_LOAD;
    accdesc.write           = memop == MemOp_STORE;

    return accdesc;
```

Library pseudocode for shared/functions/memory/CreateAccDescGCSSS1

```
// CreateAccDescGCSSS1()
// =====================
// Access descriptor for memory accesses to the Guarded Control Stack that switch stacks

AccessDescriptor CreateAccDescGCSSS1(bits(2) el)
    AccessDescriptor accdesc = NewAccDesc(AccessType_GCS);

    accdesc.el              = el;
    accdesc.atomicop        = TRUE;
    accdesc.modop           = MemAtomicOp_GCSSS1;
    accdesc.read            = TRUE;
    accdesc.write           = TRUE;

    return accdesc;
```

Library pseudocode for shared/functions/memory/CreateAccDescGPR

```
// CreateAccDescGPR()
// ==================
// Access descriptor for general purpose register loads/stores
// without exclusive or ordering semantics

AccessDescriptor CreateAccDescGPR(MemOp memop, boolean nontemporal, boolean privileged,
                                  boolean tagchecked)
    AccessDescriptor accdesc = NewAccDesc(AccessType_GPR);

    accdesc.el              = if !privileged then EL0 else PSTATE.EL;
    accdesc.nontemporal     = nontemporal;
    accdesc.read            = memop == MemOp_LOAD;
    accdesc.write           = memop == MemOp_STORE;
    accdesc.pan             = TRUE;
    accdesc.tagchecked      = tagchecked;
    accdesc.transactional   = HaveTME() && TSTATE.depth > 0;

    return accdesc;
```

Library pseudocode for shared/functions/memory/CreateAccDescGPTW

```
// CreateAccDescGPTW()
// ===================
// Access descriptor for Granule Protection Table walks

AccessDescriptor CreateAccDescGPTW(AccessDescriptor accdesc_in)
    AccessDescriptor accdesc = NewAccDesc(AccessType_GPTW);

    accdesc.el              = accdesc_in.el;
    accdesc.ss              = accdesc_in.ss;
    accdesc.read            = TRUE;
    accdesc.mpam            = accdesc_in.mpam;

    return accdesc;
```

Library pseudocode for shared/functions/memory/CreateAccDescIC

```
// CreateAccDescIC()
// =================
// Access descriptor for instruction cache operations

AccessDescriptor CreateAccDescIC(CacheRecord cache)
    AccessDescriptor accdesc = NewAccDesc(AccessType_IC);

    accdesc.cacheop         = cache.cacheop;
    accdesc.cachetype       = cache.cachetype;
    accdesc.opscope         = cache.opscope;

    return accdesc;
```

Library pseudocode for shared/functions/memory/CreateAccDescIFetch

```
// CreateAccDescIFetch()
// =====================
// Access descriptor for instruction fetches

AccessDescriptor CreateAccDescIFetch()
    AccessDescriptor accdesc = NewAccDesc(AccessType_IFETCH);

    return accdesc;
```

Library pseudocode for shared/functions/memory/CreateAccDescLDAcqPC

```
// CreateAccDescLDAcqPC()
// ======================
// Access descriptor for general purpose register loads with local ordering semantics

AccessDescriptor CreateAccDescLDAcqPC(boolean tagchecked)
    AccessDescriptor accdesc = NewAccDesc(AccessType_GPR);

    accdesc.acqpc           = TRUE;
    accdesc.read            = TRUE;
    accdesc.pan             = TRUE;
    accdesc.tagchecked      = tagchecked;
    accdesc.transactional   = HaveTME() && TSTATE.depth > 0;

    return accdesc;
```

Library pseudocode for shared/functions/memory/CreateAccDescLDGSTG

```
// CreateAccDescLDGSTG()
// =====================
// Access descriptor for tag memory loads/stores

AccessDescriptor CreateAccDescLDGSTG(MemOp memop)
    AccessDescriptor accdesc = NewAccDesc(AccessType_GPR);

    accdesc.read            = memop == MemOp_LOAD;
    accdesc.write           = memop == MemOp_STORE;
    accdesc.pan             = TRUE;
    accdesc.tagaccess       = TRUE;
    accdesc.transactional   = HaveTME() && TSTATE.depth > 0;

    return accdesc;
```

Library pseudocode for shared/functions/memory/CreateAccDescLOR

```
// CreateAccDescLOR()
// ==================
// Access descriptor for general purpose register loads/stores with limited ordering semantics

AccessDescriptor CreateAccDescLOR(MemOp memop, boolean tagchecked)
    AccessDescriptor accdesc = NewAccDesc(AccessType_GPR);

    accdesc.acqsc           = memop == MemOp_LOAD;
    accdesc.relsc           = memop == MemOp_STORE;
    accdesc.limitedordered  = TRUE;
    accdesc.read            = memop == MemOp_LOAD;
    accdesc.write           = memop == MemOp_STORE;
    accdesc.pan             = TRUE;
    accdesc.tagchecked      = tagchecked;
    accdesc.transactional   = HaveTME() && TSTATE.depth > 0;

    return accdesc;
```

Library pseudocode for shared/functions/memory/CreateAccDescLS64

```
// CreateAccDescLS64()
// ===================
// Access descriptor for accelerator-supporting memory accesses

AccessDescriptor CreateAccDescLS64(MemOp memop, boolean tagchecked)
    AccessDescriptor accdesc = NewAccDesc(AccessType_GPR);

    accdesc.read            = memop == MemOp_LOAD;
    accdesc.write           = memop == MemOp_STORE;
    accdesc.pan             = TRUE;
    accdesc.ls64            = TRUE;
    accdesc.tagchecked      = tagchecked;
    accdesc.transactional   = HaveTME() && TSTATE.depth > 0;

    return accdesc;
```

Library pseudocode for shared/functions/memory/CreateAccDescMOPS

```
// CreateAccDescMOPS()
// ===================
// Access descriptor for data memory copy and set instructions

AccessDescriptor CreateAccDescMOPS(MemOp memop, boolean privileged, boolean nontemporal)
    AccessDescriptor accdesc = NewAccDesc(AccessType_GPR);

    accdesc.el              = if !privileged then EL0 else PSTATE.EL;
    accdesc.nontemporal     = nontemporal;
    accdesc.read            = memop == MemOp_LOAD;
    accdesc.write           = memop == MemOp_STORE;
    accdesc.pan             = TRUE;
    accdesc.mops            = TRUE;
    accdesc.tagchecked      = TRUE;
    accdesc.transactional   = HaveTME() && TSTATE.depth > 0;

    return accdesc;
```

Library pseudocode for shared/functions/memory/CreateAccDescNV2

```
// CreateAccDescNV2()
// ==================
// Access descriptor nested virtualization memory indirection loads/stores

AccessDescriptor CreateAccDescNV2(MemOp memop)
    AccessDescriptor accdesc = NewAccDesc(AccessType_NV2);

    accdesc.el              = EL2;
    accdesc.ss              = SecurityStateAtEL(EL2);
    accdesc.read            = memop == MemOp_LOAD;
    accdesc.write           = memop == MemOp_STORE;
    accdesc.transactional   = HaveTME() && TSTATE.depth > 0;

    return accdesc;
```

Library pseudocode for shared/functions/memory/CreateAccDescRCW

```
// CreateAccDescRCW()
// ==================
// Access descriptor for atomic read-check-write memory accesses

AccessDescriptor CreateAccDescRCW(MemAtomicOp modop, boolean soft, boolean acquire,
                                  boolean release, boolean tagchecked)
    AccessDescriptor accdesc = NewAccDesc(AccessType_GPR);

    accdesc.acqsc           = acquire;
    accdesc.relsc           = release;
    accdesc.rcw             = TRUE;
    accdesc.rcws            = soft;
    accdesc.atomicop        = TRUE;
    accdesc.modop           = modop;
    accdesc.read            = TRUE;
    accdesc.write           = TRUE;
    accdesc.pan             = TRUE;
    accdesc.tagchecked      = tagchecked;
    accdesc.transactional   = HaveTME() && TSTATE.depth > 0;

    return accdesc;
```

Library pseudocode for shared/functions/memory/CreateAccDescS1TTW

```
// CreateAccDescS1TTW()
// ====================
// Access descriptor for stage 1 translation table walks

AccessDescriptor CreateAccDescS1TTW(boolean toplevel, VARange varange, AccessDescriptor accdesc_in)
    AccessDescriptor accdesc = NewAccDesc(AccessType_TTW);

    accdesc.el              = accdesc_in.el;
    accdesc.ss              = accdesc_in.ss;
    accdesc.read            = TRUE;
    accdesc.toplevel        = toplevel;
    accdesc.varange         = varange;
    accdesc.mpam            = accdesc_in.mpam;

    return accdesc;
```

Library pseudocode for shared/functions/memory/CreateAccDescS2TTW

```
// CreateAccDescS2TTW()
// ====================
// Access descriptor for stage 2 translation table walks

AccessDescriptor CreateAccDescS2TTW(AccessDescriptor accdesc_in)
    AccessDescriptor accdesc = NewAccDesc(AccessType_TTW);

    accdesc.el              = accdesc_in.el;
    accdesc.ss              = accdesc_in.ss;
    accdesc.read            = TRUE;
    accdesc.mpam            = accdesc_in.mpam;

    return accdesc;
```

Library pseudocode for shared/functions/memory/CreateAccDescSME

```
// CreateAccDescSME()
// ==================
// Access descriptor for SME loads/stores

AccessDescriptor CreateAccDescSME(MemOp memop, boolean nontemporal, boolean contiguous,
                                  boolean tagchecked)
    AccessDescriptor accdesc = NewAccDesc(AccessType_SME);

    accdesc.nontemporal     = nontemporal;
    accdesc.read            = memop == MemOp_LOAD;
    accdesc.write           = memop == MemOp_STORE;
    accdesc.pan             = TRUE;
    accdesc.contiguous      = contiguous;
    accdesc.streamingsve    = TRUE;
    if boolean IMPLEMENTATION_DEFINED "No tag checking of SME LDR & STR instructions" then
        accdesc.tagchecked  = FALSE;
    else
        accdesc.tagchecked  = tagchecked;
    accdesc.transactional   = HaveTME() && TSTATE.depth > 0;

    return accdesc;
```

Library pseudocode for shared/functions/memory/CreateAccDescSPE

```
// CreateAccDescSPE()
// ==================
// Access descriptor for memory accesses by Statistical Profiling unit

AccessDescriptor CreateAccDescSPE(SecurityState owning_ss, bits(2) owning_el)
    AccessDescriptor accdesc = NewAccDesc(AccessType_SPE);

    accdesc.el              = owning_el;
    accdesc.ss              = owning_ss;
    accdesc.write           = TRUE;
    accdesc.mpam            = GenMPAMatEL(AccessType_SPE, owning_el);

    return accdesc;
```

Library pseudocode for shared/functions/memory/CreateAccDescSTGMOPS

```
// CreateAccDescSTGMOPS()
// ======================
// Access descriptor for tag memory set instructions

AccessDescriptor CreateAccDescSTGMOPS(boolean privileged, boolean nontemporal)
    AccessDescriptor accdesc = NewAccDesc(AccessType_GPR);

    accdesc.el              = if !privileged then EL0 else PSTATE.EL;
    accdesc.nontemporal     = nontemporal;
    accdesc.write           = TRUE;
    accdesc.pan             = TRUE;
    accdesc.mops            = TRUE;
    accdesc.tagaccess       = TRUE;
    accdesc.transactional   = HaveTME() && TSTATE.depth > 0;

    return accdesc;
```

Library pseudocode for shared/functions/memory/CreateAccDescSVE

```
// CreateAccDescSVE()
// ==================
// Access descriptor for general SVE loads/stores

AccessDescriptor CreateAccDescSVE(MemOp memop, boolean nontemporal, boolean contiguous,
                                  boolean tagchecked)
    AccessDescriptor accdesc = NewAccDesc(AccessType_SVE);

    accdesc.nontemporal     = nontemporal;
    accdesc.read            = memop == MemOp_LOAD;
    accdesc.write           = memop == MemOp_STORE;
    accdesc.pan             = TRUE;
    accdesc.contiguous      = contiguous;
    accdesc.streamingsve    = InStreamingMode();
    if (accdesc.streamingsve && boolean IMPLEMENTATION_DEFINED
          "No tag checking of SIMD&FP loads and stores in Streaming SVE mode") then
        accdesc.tagchecked  = FALSE;
    else
        accdesc.tagchecked  = tagchecked;
    accdesc.transactional   = HaveTME() && TSTATE.depth > 0;

    return accdesc;
```

Library pseudocode for shared/functions/memory/CreateAccDescSVEFF

```
// CreateAccDescSVEFF()
// ====================
// Access descriptor for first-fault SVE loads

AccessDescriptor CreateAccDescSVEFF(boolean contiguous, boolean tagchecked)
    AccessDescriptor accdesc = NewAccDesc(AccessType_SVE);

    accdesc.read            = TRUE;
    accdesc.pan             = TRUE;
    accdesc.firstfault      = TRUE;
    accdesc.first           = TRUE;
    accdesc.contiguous      = contiguous;
    accdesc.streamingsve    = InStreamingMode();
    if (accdesc.streamingsve && boolean IMPLEMENTATION_DEFINED
          "No tag checking of SIMD&FP loads and stores in Streaming SVE mode") then
        accdesc.tagchecked  = FALSE;
    else
        accdesc.tagchecked  = tagchecked;
    accdesc.transactional   = HaveTME() && TSTATE.depth > 0;

    return accdesc;
```

Library pseudocode for shared/functions/memory/CreateAccDescSVENF

```
// CreateAccDescSVENF()
// ====================
// Access descriptor for non-fault SVE loads

AccessDescriptor CreateAccDescSVENF(boolean contiguous, boolean tagchecked)
    AccessDescriptor accdesc = NewAccDesc(AccessType_SVE);

    accdesc.read            = TRUE;
    accdesc.pan             = TRUE;
    accdesc.nonfault        = TRUE;
    accdesc.contiguous      = contiguous;
    accdesc.streamingsve    = InStreamingMode();
    if (accdesc.streamingsve && boolean IMPLEMENTATION_DEFINED
          "No tag checking of SIMD&FP loads and stores in Streaming SVE mode") then
        accdesc.tagchecked  = FALSE;
    else
        accdesc.tagchecked  = tagchecked;
    accdesc.transactional   = HaveTME() && TSTATE.depth > 0;

    return accdesc;
```

Library pseudocode for shared/functions/memory/CreateAccDescTRBE

```
// CreateAccDescTRBE()
// ===================
// Access descriptor for memory accesses by Trace Buffer Unit

AccessDescriptor CreateAccDescTRBE(SecurityState owning_ss, bits(2) owning_el)
    AccessDescriptor accdesc = NewAccDesc(AccessType_TRBE);

    accdesc.el              = owning_el;
    accdesc.ss              = owning_ss;
    accdesc.write           = TRUE;

    return accdesc;
```

Library pseudocode for shared/functions/memory/CreateAccDescTTEUpdate

```
// CreateAccDescTTEUpdate()
// ========================
// Access descriptor for translation table entry HW update

AccessDescriptor CreateAccDescTTEUpdate(AccessDescriptor accdesc_in)
    AccessDescriptor accdesc = NewAccDesc(AccessType_TTW);

    accdesc.el              = accdesc_in.el;
    accdesc.ss              = accdesc_in.ss;
    accdesc.atomicop        = TRUE;
    accdesc.modop           = MemAtomicOp_CAS;
    accdesc.read            = TRUE;
    accdesc.write           = TRUE;
    accdesc.mpam            = accdesc_in.mpam;

    return accdesc;
```

Library pseudocode for shared/functions/memory/DataMemoryBarrier

```
// DataMemoryBarrier()
// ===================

DataMemoryBarrier(MBReqDomain domain, MBReqTypes types);
```

Library pseudocode for shared/functions/memory/DataSynchronizationBarrier

```
// DataSynchronizationBarrier()
// ============================

DataSynchronizationBarrier(MBReqDomain domain, MBReqTypes types, boolean nXS);
```

Library pseudocode for shared/functions/memory/DeviceType

```
// DeviceType
// ==========
// Extended memory types for Device memory.

enumeration DeviceType {DeviceType_GRE, DeviceType_nGRE, DeviceType_nGnRE, DeviceType_nGnRnE};
```

Library pseudocode for shared/functions/memory/EffectiveMTX

```
// EffectiveMTX()
// ==============
// Returns the effective MTX in the AArch64 stage 1 translation regime for "el".

bit EffectiveMTX(bits(64) address, boolean is_instr, bits(2) el)
    bit mtx;
    assert HaveEL(el);
    regime = S1TranslationRegime(el);
    assert(!ELUsingAArch32(regime));

    if !HaveMTE4Ext() || is_instr then
        mtx = '0';
    else
        case regime of
            when EL1
                mtx = if address<55> == '1' then TCR_EL1.MTX1 else TCR_EL1.MTX0;
            when EL2
                if HaveVirtHostExt() && ELIsInHost(el) then
                    mtx = if address<55> == '1' then TCR_EL2.MTX1 else TCR_EL2.MTX0;
                else
                    mtx = TCR_EL2.MTX;
            when EL3
                mtx = TCR_EL3.MTX;

    return mtx;
```

Library pseudocode for shared/functions/memory/EffectiveTBI

```
// EffectiveTBI()
// ==============
// Returns the effective TBI in the AArch64 stage 1 translation regime for "el".

bit EffectiveTBI(bits(64) address, boolean IsInstr, bits(2) el)
    bit tbi;
    bit tbid;
    assert HaveEL(el);
    regime = S1TranslationRegime(el);
    assert(!ELUsingAArch32(regime));

    case regime of
        when EL1
            tbi = if address<55> == '1' then TCR_EL1.TBI1 else TCR_EL1.TBI0;
            if HavePACExt() then
                tbid = if address<55> == '1' then TCR_EL1.TBID1 else TCR_EL1.TBID0;
        when EL2
            if HaveVirtHostExt() && ELIsInHost(el) then
                tbi = if address<55> == '1' then TCR_EL2.TBI1 else TCR_EL2.TBI0;
                if HavePACExt() then
                    tbid = if address<55> == '1' then TCR_EL2.TBID1 else TCR_EL2.TBID0;
            else
                tbi = TCR_EL2.TBI;
                if HavePACExt() then tbid = TCR_EL2.TBID;
        when EL3
            tbi = TCR_EL3.TBI;
            if HavePACExt() then tbid = TCR_EL3.TBID;

    return (if tbi == '1' && (!HavePACExt() || tbid == '0' || !IsInstr) then '1' else '0');
```

Library pseudocode for shared/functions/memory/EffectiveTCMA

```
// EffectiveTCMA()
// ===============
// Returns the effective TCMA of a virtual address in the stage 1 translation regime for "el".

bit EffectiveTCMA(bits(64) address, bits(2) el)
    bit tcma;
    assert HaveEL(el);
    regime = S1TranslationRegime(el);
    assert(!ELUsingAArch32(regime));

    case regime of
        when EL1
            tcma = if address<55> == '1' then TCR_EL1.TCMA1 else TCR_EL1.TCMA0;
        when EL2
            if HaveVirtHostExt() && ELIsInHost(el) then
                tcma = if address<55> == '1' then TCR_EL2.TCMA1 else TCR_EL2.TCMA0;
            else
                tcma = TCR_EL2.TCMA;
        when EL3
            tcma = TCR_EL3.TCMA;

    return tcma;
```

Library pseudocode for shared/functions/memory/ErrorState

```
// ErrorState
// ==========
// The allowed error states that can be returned by memory and used by the PE.

enumeration ErrorState {ErrorState_UC,            // Uncontainable
                        ErrorState_UEU,           // Unrecoverable state
                        ErrorState_UEO,           // Restartable state
                        ErrorState_UER,           // Recoverable state
                        ErrorState_CE,            // Corrected
                        ErrorState_Uncategorized,
                        ErrorState_IMPDEF};
```

Library pseudocode for shared/functions/memory/Fault

```
// Fault
// =====
// Fault types.

enumeration Fault {Fault_None,
                   Fault_AccessFlag,
                   Fault_Alignment,
                   Fault_Background,
                   Fault_Domain,
                   Fault_Permission,
                   Fault_Translation,
                   Fault_AddressSize,
                   Fault_SyncExternal,
                   Fault_SyncExternalOnWalk,
                   Fault_SyncParity,
                   Fault_SyncParityOnWalk,
                   Fault_GPCFOnWalk,
                   Fault_GPCFOnOutput,
                   Fault_AsyncParity,
                   Fault_AsyncExternal,
                   Fault_TagCheck,
                   Fault_Debug,
                   Fault_TLBConflict,
                   Fault_BranchTarget,
                   Fault_HWUpdateAccessFlag,
                   Fault_Lockdown,
                   Fault_Exclusive,
                   Fault_ICacheMaint};
```

Library pseudocode for shared/functions/memory/FaultRecord

```
// FaultRecord
// ===========
// Fields that relate only to Faults.

type FaultRecord is (
    Fault             statuscode,  // Fault Status
    AccessDescriptor access,      // Details of the faulting access
    FullAddress      ipaddress,   // Intermediate physical address
    GPCFRecord       gpcf,        // Granule Protection Check Fault record
    FullAddress      paddress,    // Physical address
    boolean          gpcfs2walk,  // GPC for a stage 2 translation table walk
    boolean          s2fs1walk,   // Is on a Stage 1 translation table walk
    boolean          write,       // TRUE for a write, FALSE for a read
    boolean          s1tagnotdata,// TRUE for a fault due to tag not accessible at stage 1.
    boolean          tagaccess,   // TRUE for a fault due to NoTagAccess permission.
    integer          level,       // For translation, access flag and Permission faults
    bit              extflag,     // IMPLEMENTATION DEFINED syndrome for External aborts
    boolean          secondstage, // Is a Stage 2 abort
    boolean          assuredonly, // Stage 2 Permission fault due to AssuredOnly attribute
    boolean          toplevel,    // Stage 2 Permission fault due to TopLevel
    boolean          overlay,     // Fault due to overlay permissions
    boolean          dirtybit,    // Fault due to dirty state
    bits(4)          domain,      // Domain number, AArch32 only
    ErrorState       merrorstate, // Incoming error state from memory
    bits(4)          debugmoe     // Debug method of entry, from AArch32 only
)
```

Library pseudocode for shared/functions/memory/FullAddress

```
// FullAddress
// ===========
// Physical or Intermediate Physical Address type.
// Although AArch32 only has access to 40 bits of physical or intermediate physical address space,
// the full address type has 56 bits to allow interprocessing with AArch64.
// The maximum physical or intermediate physical address size is IMPLEMENTATION DEFINED,
// but never exceeds 56 bits.

type FullAddress is (
    PASpace  paspace,
    bits(56) address
)
```

Library pseudocode for shared/functions/memory/GPCF

```
// GPCF
// ====
// Possible Granule Protection Check Fault reasons

enumeration GPCF {
    GPCF_None,        // No fault
    GPCF_AddressSize, // GPT address size fault
    GPCF_Walk,        // GPT walk fault
    GPCF_EABT,        // Synchronous External abort on GPT fetch
    GPCF_Fail         // Granule protection fault
};
```

Library pseudocode for shared/functions/memory/GPCFRecord

```
// GPCFRecord
// ==========
// Full details of a Granule Protection Check Fault

type GPCFRecord is (
    GPCF    gpf,
    integer level
)
```

Library pseudocode for shared/functions/memory/Hint_Prefetch

```
// Hint_Prefetch()
// ===============
// Signals the memory system that memory accesses of type HINT to or from the specified address are
// likely in the near future. The memory system may take some action to speed up the memory
// accesses when they do occur, such as pre-loading the specified address into one or more
// caches as indicated by the innermost cache level target (0=L1, 1=L2, etc) and non-temporal hint
// stream. Any or all prefetch hints may be treated as a NOP. A prefetch hint must not cause a
// synchronous abort due to Alignment or Translation faults and the like. Its only effect on
// software-visible state should be on caches and TLBs associated with address, which must be
// accessible by reads, writes or execution, as defined in the translation regime of the current
// Exception level. It is guaranteed not to access Device memory.
// A Prefetch_EXEC hint must not result in an access that could not be performed by a speculative
// instruction fetch, therefore if all associated MMUs are disabled, then it cannot access any
// memory location that cannot be accessed by instruction fetches.

Hint_Prefetch(bits(64) address, PrefetchHint hint, integer target, boolean stream);
```

Library pseudocode for shared/functions/memory/Hint_RangePrefetch

```
// Hint_RangePrefetch()
// ====================
// Signals the memory system that data memory accesses from a specified range
// of addresses are likely to occur in the near future. The memory system can
// respond by taking actions that are expected to speed up the memory accesses
// when they do occur, such as preloading the locations within the specified
// address ranges into one or more caches.

Hint_RangePrefetch(bits(64) address, integer length, integer stride,
                   integer count, integer reuse, bits(6) operation);
```

Library pseudocode for shared/functions/memory/IsDataAccess

```
// IsDataAccess()
// ==============
// Return TRUE if access is to data memory.

boolean IsDataAccess(AccessType acctype)
    return !(acctype IN {AccessType_IFETCH,
                         AccessType_TTW,
                         AccessType_DC,
                         AccessType_IC,
                         AccessType_AT});
```

Library pseudocode for shared/functions/memory/MBReqDomain

```
// MBReqDomain
// ===========
// Memory barrier domain.

enumeration MBReqDomain    {MBReqDomain_Nonshareable, MBReqDomain_InnerShareable,
                            MBReqDomain_OuterShareable, MBReqDomain_FullSystem};
```

Library pseudocode for shared/functions/memory/MBReqTypes

```
// MBReqTypes
// ==========
// Memory barrier read/write.

enumeration MBReqTypes     {MBReqTypes_Reads, MBReqTypes_Writes, MBReqTypes_All};
```

Library pseudocode for shared/functions/memory/MPAM

```
// MPAM Types
// ==========

type PARTIDtype = bits(16);

type PMGtype = bits(8);

enumeration PARTIDspaceType {
    PIdSpace_Secure,
    PIdSpace_Root,
    PIdSpace_Realm,
    PIdSpace_NonSecure
};

type MPAMinfo is (
     PARTIDspaceType mpam_sp,
     PARTIDtype partid,
     PMGtype pmg
)
```

Library pseudocode for shared/functions/memory/MemAtomicOp

```
// MemAtomicOp
// ===========
// Atomic data processing instruction types.

enumeration MemAtomicOp {
    MemAtomicOp_GCSSS1,
    MemAtomicOp_ADD,
    MemAtomicOp_BIC,
    MemAtomicOp_EOR,
    MemAtomicOp_ORR,
    MemAtomicOp_SMAX,
    MemAtomicOp_SMIN,
    MemAtomicOp_UMAX,
    MemAtomicOp_UMIN,
    MemAtomicOp_SWP,
    MemAtomicOp_CAS
};

enumeration CacheOp {
    CacheOp_Clean,
    CacheOp_Invalidate,
    CacheOp_CleanInvalidate
};

enumeration CacheOpScope {
    CacheOpScope_SetWay,
    CacheOpScope_PoU,
    CacheOpScope_PoC,
    CacheOpScope_PoE,
    CacheOpScope_PoP,
    CacheOpScope_PoDP,
    CacheOpScope_PoPA,
    CacheOpScope_ALLU,
    CacheOpScope_ALLUIS
};

enumeration CacheType {
    CacheType_Data,
    CacheType_Tag,
    CacheType_Data_Tag,
    CacheType_Instruction
};

enumeration CachePASpace {
    CPAS_NonSecure,
    CPAS_Any,               // Applicable only for DC *SW / IC IALLU* in Root state:
                            // match entries from any PA Space
    CPAS_RealmNonSecure,    // Applicable only for DC *SW / IC IALLU* in Realm state:
                            // match entries from Realm or Non-Secure PAS
    CPAS_Realm,
    CPAS_Root,
    CPAS_SecureNonSecure,   // Applicable only for DC *SW / IC IALLU* in Secure state:
                            // match entries from Secure or Non-Secure PAS
    CPAS_Secure
};
```

Library pseudocode for shared/functions/memory/MemAttrHints

```
// MemAttrHints
// ============
// Attributes and hints for Normal memory.

type MemAttrHints is (
    bits(2) attrs,  // See MemAttr_*, Cacheability attributes
    bits(2) hints,  // See MemHint_*, Allocation hints
    boolean transient
)
```

Library pseudocode for shared/functions/memory/MemOp

```
// MemOp
// =====
// Memory access instruction types.

enumeration MemOp {MemOp_LOAD, MemOp_STORE, MemOp_PREFETCH};
```

Library pseudocode for shared/functions/memory/MemType

```
// MemType
// =======
// Basic memory types.

enumeration MemType {MemType_Normal, MemType_Device};
```

Library pseudocode for shared/functions/memory/Memory

```
// Memory Tag type
// ===============

enumeration MemTagType {
    MemTag_Untagged,
    MemTag_AllocationTagged,
    MemTag_CanonicallyTagged
};
```

Library pseudocode for shared/functions/memory/MemoryAttributes

```
// MemoryAttributes
// ================
// Memory attributes descriptor

type MemoryAttributes is (
    MemType      memtype,
    DeviceType   device,        // For Device memory types
    MemAttrHints inner,         // Inner hints and attributes
    MemAttrHints outer,         // Outer hints and attributes
    Shareability shareability, // Shareability attribute
    MemTagType   tags,          // MTE tag type for this memory.
    boolean      notagaccess,  // Allocation Tag access permission
    bit          xs             // XS attribute
)
```

Library pseudocode for shared/functions/memory/NewAccDesc

```
// NewAccDesc()
// ============
// Create a new AccessDescriptor with initialised fields

AccessDescriptor NewAccDesc(AccessType acctype)
    AccessDescriptor accdesc;

    accdesc.acctype         = acctype;
    accdesc.el              = PSTATE.EL;
    accdesc.ss              = SecurityStateAtEL(PSTATE.EL);
    accdesc.acqsc           = FALSE;
    accdesc.acqpc           = FALSE;
    accdesc.relsc           = FALSE;
    accdesc.limitedordered  = FALSE;
    accdesc.exclusive       = FALSE;
    accdesc.rcw             = FALSE;
    accdesc.rcws            = FALSE;
    accdesc.atomicop        = FALSE;
    accdesc.nontemporal     = FALSE;
    accdesc.read            = FALSE;
    accdesc.write           = FALSE;
    accdesc.pan             = FALSE;
    accdesc.nonfault        = FALSE;
    accdesc.firstfault      = FALSE;
    accdesc.first           = FALSE;
    accdesc.contiguous      = FALSE;
    accdesc.streamingsve    = FALSE;
    accdesc.ls64            = FALSE;
    accdesc.mops            = FALSE;
    accdesc.a32lsmd         = FALSE;
    accdesc.tagchecked      = FALSE;
    accdesc.tagaccess       = FALSE;
    accdesc.transactional   = FALSE;
    accdesc.mpam            = GenMPAMcurEL(acctype);

    return accdesc;
```

Library pseudocode for shared/functions/memory/PASpace

```
// PASpace
// =======
// Physical address spaces

enumeration PASpace {
    PAS_NonSecure,
    PAS_Secure,
    PAS_Root,
    PAS_Realm
};
```

Library pseudocode for shared/functions/memory/Permissions

```
// Permissions
// ===========
// Access Control bits in translation table descriptors

type Permissions is (
    bits(2) ap_table,   // Stage 1 hierarchical access permissions
    bit     xn_table,   // Stage 1 hierarchical execute-never for single EL regimes
    bit     pxn_table,  // Stage 1 hierarchical privileged execute-never
    bit     uxn_table,  // Stage 1 hierarchical unprivileged execute-never
    bits(3) ap,         // Stage 1 access permissions
    bit     xn,         // Stage 1 execute-never for single EL regimes
    bit     uxn,        // Stage 1 unprivileged execute-never
    bit     pxn,        // Stage 1 privileged execute-never
    bits(4) ppi,        // Stage 1 privileged indirect permissions
    bits(4) upi,        // Stage 1 unprivileged indirect permissions
    bit     ndirty,     // Stage 1 dirty state for indirect permissions scheme
    bits(4) s2pi,       // Stage 2 indirect permissions
    bit     s2dirty,    // Stage 2 dirty state
    bits(4) po_index,   // Stage 1 overlay permissions index
    bits(4) s2po_index, // Stage 2 overlay permissions index
    bits(2) s2ap,       // Stage 2 access permissions
    bit     s2tag_na,   // Stage 2 tag access
    bit     s2xnx,      // Stage 2 extended execute-never
    bit     s2xn        // Stage 2 execute-never
)
```

Library pseudocode for shared/functions/memory/PhysMemRead

```
// PhysMemRead()
// =============
// Returns the value read from memory, and a status.
// Returned value is UNKNOWN if an External abort occurred while reading the
// memory.
// Otherwise the PhysMemRetStatus statuscode is Fault_None.

(PhysMemRetStatus, bits(8*size)) PhysMemRead(AddressDescriptor desc, integer size,
                                             AccessDescriptor accdesc);
```

Library pseudocode for shared/functions/memory/PhysMemRetStatus

```
// PhysMemRetStatus
// ================
// Fields that relate only to return values of PhysMem functions.

type PhysMemRetStatus is (
    Fault       statuscode,     // Fault Status
    bit         extflag,        // IMPLEMENTATION DEFINED syndrome for External aborts
    ErrorState  merrorstate,    // Optional error state returned on a physical memory access
    bits(64)    store64bstatus  // Status of 64B store
)
```

Library pseudocode for shared/functions/memory/PhysMemWrite

```
// PhysMemWrite()
// ==============
// Writes the value to memory, and returns the status of the write.
// If there is an External abort on the write, the PhysMemRetStatus indicates this.
// Otherwise the statuscode of PhysMemRetStatus is Fault_None.

PhysMemRetStatus PhysMemWrite(AddressDescriptor desc, integer size, AccessDescriptor accdesc,
                              bits(8*size) value);
```

Library pseudocode for shared/functions/memory/PrefetchHint

```
// PrefetchHint
// ============
// Prefetch hint types.

enumeration PrefetchHint {Prefetch_READ, Prefetch_WRITE, Prefetch_EXEC};
```

Library pseudocode for shared/functions/memory/S1AccessControls

```
// S1AccessControls
// ================
// Effective access controls defined by stage 1 translation

type S1AccessControls is (
    bit r,                  // Stage 1 base read permission
    bit w,                  // Stage 1 base write permission
    bit x,                  // Stage 1 base execute permission
    bit gcs,                // Stage 1 GCS permission
    boolean overlay,        // Stage 1 overlay feature enabled
    bit or,                 // Stage 1 overlay read permission
    bit ow,                 // Stage 1 overlay write permission
    bit ox,                 // Stage 1 overlay execute permission
    bit wxn                 // Stage 1 write permission implies execute-never
)
```

Library pseudocode for shared/functions/memory/S2AccessControls

```
// S2AccessControls
// ================
// Effective access controls defined by stage 2 translation

type S2AccessControls is (
    bit r,                  // Stage 2 read permission.
    bit w,                  // Stage 2 write permission.
    bit x,                  // Stage 2 execute permission.
    bit r_rcw,              // Stage 2 Read perms for RCW instruction.
    bit w_rcw,              // Stage 2 Write perms for RCW instruction.
    bit r_mmu,              // Stage 2 Read perms for TTW data.
    bit w_mmu,              // Stage 2 Write perms for TTW data.
    bit toplevel0,          // IPA as top level table for TTBR0_EL1.
    bit toplevel1,          // IPA as top level table for TTBR1_EL1.
    boolean overlay,        // Overlay enable
    bit or,                 // Stage 2 overlay read permission.
    bit ow,                 // Stage 2 overlay write permission.
    bit ox,                 // Stage 2 overlay execute permission.
    bit or_rcw,             // Stage 2 overlay Read perms for RCW instruction.
    bit ow_rcw,             // Stage 2 overlay Write perms for RCW instruction.
    bit or_mmu,             // Stage 2 overlay Read perms for TTW data.
    bit ow_mmu,             // Stage 2 overlay Write perms for TTW data.
)
```

Library pseudocode for shared/functions/memory/Shareability

```
// Shareability
// ============

enumeration Shareability {
    Shareability_NSH,
    Shareability_ISH,
    Shareability_OSH
};
```

Library pseudocode for shared/functions/memory/SpeculativeStoreBypassBarrierToPA

```
// SpeculativeStoreBypassBarrierToPA()
// ===================================

SpeculativeStoreBypassBarrierToPA();
```

Library pseudocode for shared/functions/memory/SpeculativeStoreBypassBarrierToVA

```
// SpeculativeStoreBypassBarrierToVA()
// ===================================

SpeculativeStoreBypassBarrierToVA();
```

Library pseudocode for shared/functions/memory/Tag

```
constant integer LOG2_TAG_GRANULE = 4;

constant integer TAG_GRANULE = 1 << LOG2_TAG_GRANULE;
```

Library pseudocode for shared/functions/memory/VARange

```
// VARange
// =======
// Virtual address ranges

enumeration VARange {
    VARange_LOWER,
    VARange_UPPER
};
```

Library pseudocode for shared/functions/mpam/AltPARTIDspace

```
// AltPARTIDspace()
// ================
// From the Security state, EL and ALTSP configuration, determine
// whether to primary space or the alt space is selected and which
// PARTID space is the alternative space. Return that alternative
// PARTID space if selected or the primary space if not.

PARTIDspaceType AltPARTIDspace(bits(2) el, SecurityState security,
                                PARTIDspaceType primaryPIdSpace)
    case security of
        when SS_NonSecure
            assert el != EL3;
            return primaryPIdSpace; // there is no ALTSP for Non_secure
        when SS_Secure
            assert el != EL3;
            if primaryPIdSpace == PIdSpace_NonSecure then
                return primaryPIdSpace;
            return AltPIdSecure(el, primaryPIdSpace);
        when SS_Root
            assert el == EL3;
            if MPAM3_EL3.ALTSP_EL3 == '1' then
                if MPAM3_EL3.RT_ALTSP_NS == '1' then
                    return PIdSpace_NonSecure;
                else
                    return PIdSpace_Secure;
            else
                return primaryPIdSpace;
        when SS_Realm
            assert el != EL3;
            return AltPIdRealm(el, primaryPIdSpace);
        otherwise
            Unreachable();
```

Library pseudocode for shared/functions/mpam/AltPIdRealm

```
// AltPIdRealm()
// =============
// Compute PARTID space as either the primary PARTID space or
// alternative PARTID space in the Realm Security state.
// Helper for AltPARTIDspace.

PARTIDspaceType AltPIdRealm(bits(2) el, PARTIDspaceType primaryPIdSpace)
    PARTIDspaceType PIdSpace = primaryPIdSpace;
    case el of
        when EL0
            if ELIsInHost(EL0) then
                if !UsePrimarySpaceEL2() then
                    PIdSpace = PIdSpace_NonSecure;
            elsif !UsePrimarySpaceEL10() then
                PIdSpace = PIdSpace_NonSecure;
        when EL1
            if !UsePrimarySpaceEL10() then
                PIdSpace = PIdSpace_NonSecure;
        when EL2
            if !UsePrimarySpaceEL2() then
                PIdSpace = PIdSpace_NonSecure;
        otherwise
            Unreachable();
    return PIdSpace;
```

Library pseudocode for shared/functions/mpam/AltPIdSecure

```
// AltPIdSecure()
// ==============
// Compute PARTID space as either the primary PARTID space or
// alternative PARTID space in the Secure Security state.
// Helper for AltPARTIDspace.

PARTIDspaceType AltPIdSecure(bits(2) el, PARTIDspaceType primaryPIdSpace)
    PARTIDspaceType PIdSpace = primaryPIdSpace;
    boolean el2en = EL2Enabled();
    case el of
        when EL0
            if el2en then
                if ELIsInHost(EL0) then
                    if !UsePrimarySpaceEL2() then
                        PIdSpace = PIdSpace_NonSecure;
                elsif !UsePrimarySpaceEL10() then
                    PIdSpace = PIdSpace_NonSecure;
            elsif MPAM3_EL3.ALTSP_HEN == '0' && MPAM3_EL3.ALTSP_HFC == '1' then
                PIdSpace = PIdSpace_NonSecure;
        when EL1
            if el2en then
                if !UsePrimarySpaceEL10() then
                    PIdSpace = PIdSpace_NonSecure;
            elsif MPAM3_EL3.ALTSP_HEN == '0' && MPAM3_EL3.ALTSP_HFC == '1' then
                PIdSpace = PIdSpace_NonSecure;
        when EL2
            if !UsePrimarySpaceEL2() then
                PIdSpace = PIdSpace_NonSecure;
        otherwise
            Unreachable();
    return PIdSpace;
```

Library pseudocode for shared/functions/mpam/DefaultMPAMinfo

```
// DefaultMPAMinfo()
// =================
// Returns default MPAM info.  The partidspace argument sets
// the PARTID space of the default MPAM information returned.

MPAMinfo DefaultMPAMinfo(PARTIDspaceType partidspace)
    MPAMinfo DefaultInfo;
    DefaultInfo.mpam_sp = partidspace;
    DefaultInfo.partid  = DefaultPARTID;
    DefaultInfo.pmg     = DefaultPMG;
    return DefaultInfo;
```

Library pseudocode for shared/functions/mpam/DefaultPARTID

```
constant PARTIDtype DefaultPARTID = 0<15:0>;
```

Library pseudocode for shared/functions/mpam/DefaultPMG

```
constant PMGtype    DefaultPMG = 0<7:0>;
```

Library pseudocode for shared/functions/mpam/GenMPAMatEL

```
// GenMPAMatEL()
// =============
// Returns MPAMinfo for the specified EL.
// May be called if MPAM is not implemented (but in an version that supports
// MPAM), MPAM is disabled, or in AArch32.  In AArch32, convert the mode to
// EL if can and use that to drive MPAM information generation.  If mode
// cannot be converted, MPAM is not implemented, or MPAM is disabled return
// default MPAM information for the current security state.

MPAMinfo GenMPAMatEL(AccessType acctype, bits(2) el)
    bits(2) mpamEL;
    boolean validEL = FALSE;
    SecurityState security = SecurityStateAtEL(el);
    boolean InD = FALSE;
    boolean InSM = FALSE;
    PARTIDspaceType pspace = PARTIDspaceFromSS(security);
    if pspace == PIdSpace_NonSecure && !MPAMisEnabled() then
        return DefaultMPAMinfo(pspace);
    if UsingAArch32() then
        (validEL, mpamEL) = ELFromM32(PSTATE.M);
    else
        mpamEL = if acctype == AccessType_NV2 then EL2 else el;
        validEL = TRUE;
    case acctype of
        when AccessType_IFETCH, AccessType_IC
            InD = TRUE;
        when AccessType_SME
            InSM = (boolean IMPLEMENTATION_DEFINED "Shared SMCU" ||
                    boolean IMPLEMENTATION_DEFINED "MPAMSM_EL1 label precedence");
        when AccessType_ASIMD
            InSM = (HaveSME() && PSTATE.SM == '1' &&
                    (boolean IMPLEMENTATION_DEFINED "Shared SMCU" ||
                    boolean IMPLEMENTATION_DEFINED "MPAMSM_EL1 label precedence"));
        when AccessType_SVE
            InSM = (HaveSME() && PSTATE.SM == '1' &&
                    (boolean IMPLEMENTATION_DEFINED "Shared SMCU" ||
                    boolean IMPLEMENTATION_DEFINED "MPAMSM_EL1 label precedence"));
        otherwise
            // Other access types are DATA accesses
            InD = FALSE;
    if !validEL then
        return DefaultMPAMinfo(pspace);
    elsif HaveRME() && MPAMIDR_EL1.HAS_ALTSP == '1' then
        // Substitute alternative PARTID space if selected
        pspace = AltPARTIDspace(mpamEL, security, pspace);
    if HaveMPAMv0p1Ext() && MPAMIDR_EL1.HAS_FORCE_NS == '1' then
        if MPAM3_EL3.FORCE_NS == '1' && security == SS_Secure then
            pspace = PIdSpace_NonSecure;
    if (HaveMPAMv0p1Ext() || HaveMPAMv1p1Ext()) && MPAMIDR_EL1.HAS_SDEFLT == '1' then
        if MPAM3_EL3.SDEFLT == '1' && security == SS_Secure then
            return DefaultMPAMinfo(pspace);
    if !MPAMisEnabled() then
        return DefaultMPAMinfo(pspace);
    else
        return genMPAM(mpamEL, InD, InSM, pspace);
```

Library pseudocode for shared/functions/mpam/GenMPAMcurEL

```
// GenMPAMcurEL()
// ==============
// Returns MPAMinfo for the current EL and security state.
// May be called if MPAM is not implemented (but in an version that supports
// MPAM), MPAM is disabled, or in AArch32.  In AArch32, convert the mode to
// EL if can and use that to drive MPAM information generation.  If mode
// cannot be converted, MPAM is not implemented, or MPAM is disabled return
// default MPAM information for the current security state.

MPAMinfo GenMPAMcurEL(AccessType acctype)
    return GenMPAMatEL(acctype, PSTATE.EL);
```

Library pseudocode for shared/functions/mpam/MAP_vPARTID

```
// MAP_vPARTID()
// =============
// Performs conversion of virtual PARTID into physical PARTID
// Contains all of the error checking and implementation
// choices for the conversion.

(PARTIDtype, boolean) MAP_vPARTID(PARTIDtype vpartid)
    // should not ever be called if EL2 is not implemented
    // or is implemented but not enabled in the current
    // security state.
    PARTIDtype ret;
    boolean err;
    integer virt    = UInt(vpartid);
    integer vpmrmax = UInt(MPAMIDR_EL1.VPMR_MAX);

    // vpartid_max is largest vpartid supported
    integer vpartid_max = (vpmrmax << 2) + 3;

    // One of many ways to reduce vpartid to value less than vpartid_max.
    if UInt(vpartid) > vpartid_max then
        virt = virt MOD (vpartid_max+1);

    // Check for valid mapping entry.
    if MPAMVPMV_EL2<virt> == '1' then
        // vpartid has a valid mapping so access the map.
        ret = mapvpmw(virt);
        err = FALSE;

    // Is the default virtual PARTID valid?
    elsif MPAMVPMV_EL2<0> == '1' then
        // Yes, so use default mapping for vpartid == 0.
        ret = MPAMVPM0_EL2<0 +: 16>;
        err = FALSE;

    // Neither is valid so use default physical PARTID.
    else
        ret = DefaultPARTID;
        err = TRUE;

    // Check that the physical PARTID is in-range.
    // This physical PARTID came from a virtual mapping entry.
    integer partid_max = UInt(MPAMIDR_EL1.PARTID_MAX);
    if UInt(ret) > partid_max then
        // Out of range, so return default physical PARTID
        ret = DefaultPARTID;
        err = TRUE;
    return (ret, err);
```

Library pseudocode for shared/functions/mpam/MPAMisEnabled

```
// MPAMisEnabled()
// ===============
// Returns TRUE if MPAMisEnabled.

boolean MPAMisEnabled()
    el = HighestEL();
    case el of
        when EL3 return MPAM3_EL3.MPAMEN == '1';
        when EL2 return MPAM2_EL2.MPAMEN == '1';
        when EL1 return MPAM1_EL1.MPAMEN == '1';
```

Library pseudocode for shared/functions/mpam/MPAMisVirtual

```
// MPAMisVirtual()
// ===============
// Returns TRUE if MPAM is configured to be virtual at EL.

boolean MPAMisVirtual(bits(2) el)
    return (MPAMIDR_EL1.HAS_HCR == '1' && EL2Enabled() &&
            ((el == EL0 && MPAMHCR_EL2.EL0_VPMEN == '1' &&
               (HCR_EL2.E2H == '0' || HCR_EL2.TGE == '0')) ||
             (el == EL1 && MPAMHCR_EL2.EL1_VPMEN == '1')));
```

Library pseudocode for shared/functions/mpam/PARTIDspaceFromSS

```
// PARTIDspaceFromSS()
// ===================
// Returns the primary PARTID space from the Security State.

PARTIDspaceType PARTIDspaceFromSS(SecurityState security)
    case security of
        when SS_NonSecure
            return PIdSpace_NonSecure;
        when SS_Root
            return PIdSpace_Root;
        when SS_Realm
            return PIdSpace_Realm;
        when SS_Secure
            return PIdSpace_Secure;
        otherwise
            Unreachable();
```

Library pseudocode for shared/functions/mpam/UsePrimarySpaceEL10

```
// UsePrimarySpaceEL10()
// =====================
// Checks whether Primary space is configured in the
// MPAM3_EL3 and MPAM2_EL2 ALTSP control bits that affect
// MPAM ALTSP use at EL1 and EL0.

boolean UsePrimarySpaceEL10()
    if MPAM3_EL3.ALTSP_HEN == '0' then
        return MPAM3_EL3.ALTSP_HFC == '0';
    return !MPAMisEnabled() || !EL2Enabled() || MPAM2_EL2.ALTSP_HFC == '0';
```

Library pseudocode for shared/functions/mpam/UsePrimarySpaceEL2

```
// UsePrimarySpaceEL2()
// ====================
// Checks whether Primary space is configured in the
// MPAM3_EL3 and MPAM2_EL2 ALTSP control bits that affect
// MPAM ALTSP use at EL2.

boolean UsePrimarySpaceEL2()
    if MPAM3_EL3.ALTSP_HEN == '0' then
        return MPAM3_EL3.ALTSP_HFC == '0';
    return !MPAMisEnabled() || MPAM2_EL2.ALTSP_EL2 == '0';
```

Library pseudocode for shared/functions/mpam/genMPAM

```
// genMPAM()
// =========
// Returns MPAMinfo for exception level el.
// If InD is TRUE returns MPAM information using PARTID_I and PMG_I fields
// of MPAMel_ELx register and otherwise using PARTID_D and PMG_D fields.
// If InSM is TRUE returns MPAM information using PARTID_D and PMG_D fields
// of MPAMSM_EL1 register.
// Produces a PARTID in PARTID space pspace.

MPAMinfo genMPAM(bits(2) el, boolean InD, boolean InSM, PARTIDspaceType pspace)
    MPAMinfo returninfo;
    PARTIDtype partidel;
    boolean perr;
    // gstplk is guest OS application locked by the EL2 hypervisor to
    // only use EL1 the virtual machine's PARTIDs.
    boolean gstplk = (el == EL0 && EL2Enabled() &&
                      MPAMHCR_EL2.GSTAPP_PLK == '1' &&
                      HCR_EL2.TGE == '0');
    bits(2) eff_el = if gstplk then EL1 else el;
    (partidel, perr) = genPARTID(eff_el, InD, InSM);
    PMGtype groupel  = genPMG(eff_el, InD, InSM, perr);
    returninfo.mpam_sp = pspace;
    returninfo.partid  = partidel;
    returninfo.pmg     = groupel;
    return returninfo;
```

Library pseudocode for shared/functions/mpam/genPARTID

```
// genPARTID()
// ===========
// Returns physical PARTID and error boolean for exception level el.
// If InD is TRUE then PARTID is from MPAMel_ELx.PARTID_I and
// otherwise from MPAMel_ELx.PARTID_D.
// If InSM is TRUE then PARTID is from MPAMSM_EL1.PARTID_D.

(PARTIDtype, boolean) genPARTID(bits(2) el, boolean InD, boolean InSM)
    PARTIDtype partidel = getMPAM_PARTID(el, InD, InSM);
    PARTIDtype partid_max = MPAMIDR_EL1.PARTID_MAX;
    if UInt(partidel) > UInt(partid_max) then
        return (DefaultPARTID, TRUE);
    if MPAMisVirtual(el) then
        return MAP_vPARTID(partidel);
    else
        return (partidel, FALSE);
```

Library pseudocode for shared/functions/mpam/genPMG

```
// genPMG()
// ========
// Returns PMG for exception level el and I- or D-side (InD).
// If PARTID generation (genPARTID) encountered an error, genPMG() should be
// called with partid_err as TRUE.

PMGtype genPMG(bits(2) el, boolean InD, boolean InSM, boolean partid_err)
    integer pmg_max = UInt(MPAMIDR_EL1.PMG_MAX);
    // It is CONSTRAINED UNPREDICTABLE whether partid_err forces PMG to
    // use the default or if it uses the PMG from getMPAM_PMG.
    if partid_err then
        return DefaultPMG;
    PMGtype groupel = getMPAM_PMG(el, InD, InSM);
    if UInt(groupel) <= pmg_max then
        return groupel;
    return DefaultPMG;
```

Library pseudocode for shared/functions/mpam/getMPAM_PARTID

```
// getMPAM_PARTID()
// ================
// Returns a PARTID from one of the MPAMn_ELx or MPAMSM_EL1 registers.
// If InSM is TRUE, the MPAMSM_EL1 register is used. Otherwise,
// MPAMn selects the MPAMn_ELx register used.
// If InD is TRUE, selects the PARTID_I field of that
// register.  Otherwise, selects the PARTID_D field.

PARTIDtype getMPAM_PARTID(bits(2) MPAMn, boolean InD, boolean InSM)
    PARTIDtype partid;
    boolean el2avail = EL2Enabled();

    if InSM then
        partid = MPAMSM_EL1.PARTID_D;
        return partid;

    if InD then
        case MPAMn of
            when '11' partid = MPAM3_EL3.PARTID_I;
            when '10' partid = if el2avail then MPAM2_EL2.PARTID_I else Zeros(16);
            when '01' partid = MPAM1_EL1.PARTID_I;
            when '00' partid = MPAM0_EL1.PARTID_I;
            otherwise partid = PARTIDtype UNKNOWN;
    else
        case MPAMn of
            when '11' partid = MPAM3_EL3.PARTID_D;
            when '10' partid = if el2avail then MPAM2_EL2.PARTID_D else Zeros(16);
            when '01' partid = MPAM1_EL1.PARTID_D;
            when '00' partid = MPAM0_EL1.PARTID_D;
            otherwise partid = PARTIDtype UNKNOWN;
    return partid;
```

Library pseudocode for shared/functions/mpam/getMPAM_PMG

```
// getMPAM_PMG()
// =============
// Returns a PMG from one of the MPAMn_ELx or MPAMSM_EL1 registers.
// If InSM is TRUE, the MPAMSM_EL1 register is used. Otherwise,
// MPAMn selects the MPAMn_ELx register used.
// If InD is TRUE, selects the PMG_I field of that
// register.  Otherwise, selects the PMG_D field.

PMGtype getMPAM_PMG(bits(2) MPAMn, boolean InD, boolean InSM)
    PMGtype pmg;
    boolean el2avail = EL2Enabled();

    if InSM then
        pmg = MPAMSM_EL1.PMG_D;
        return pmg;

    if InD then
        case MPAMn of
            when '11' pmg = MPAM3_EL3.PMG_I;
            when '10' pmg = if el2avail then MPAM2_EL2.PMG_I else Zeros(8);
            when '01' pmg = MPAM1_EL1.PMG_I;
            when '00' pmg = MPAM0_EL1.PMG_I;
            otherwise pmg = PMGtype UNKNOWN;
    else
        case MPAMn of
            when '11' pmg = MPAM3_EL3.PMG_D;
            when '10' pmg = if el2avail then MPAM2_EL2.PMG_D else Zeros(8);
            when '01' pmg = MPAM1_EL1.PMG_D;
            when '00' pmg = MPAM0_EL1.PMG_D;
            otherwise pmg = PMGtype UNKNOWN;
    return pmg;
```

Library pseudocode for shared/functions/mpam/mapvpmw

```
// mapvpmw()
// =========
// Map a virtual PARTID into a physical PARTID using
// the MPAMVPMn_EL2 registers.
// vpartid is now assumed in-range and valid (checked by caller)
// returns physical PARTID from mapping entry.

PARTIDtype mapvpmw(integer vpartid)
    bits(64) vpmw;
    integer  wd = vpartid DIV 4;
    case wd of
        when 0 vpmw = MPAMVPM0_EL2;
        when 1 vpmw = MPAMVPM1_EL2;
        when 2 vpmw = MPAMVPM2_EL2;
        when 3 vpmw = MPAMVPM3_EL2;
        when 4 vpmw = MPAMVPM4_EL2;
        when 5 vpmw = MPAMVPM5_EL2;
        when 6 vpmw = MPAMVPM6_EL2;
        when 7 vpmw = MPAMVPM7_EL2;
        otherwise vpmw = Zeros(64);
    // vpme_lsb selects LSB of field within register
    integer vpme_lsb = (vpartid MOD 4) * 16;
    return vpmw<vpme_lsb +: 16>;
```

Library pseudocode for shared/functions/predictionrestrict/ASID

```
// ASID[]
// ======
// Effective ASID.

bits(16) ASID[]
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H, TGE> == '11' then
        if TCR_EL2.A1 == '1' then
            return TTBR1_EL2.ASID;
        else
            return TTBR0_EL2.ASID;

    elsif !ELUsingAArch32(EL1) then
        if TCR_EL1.A1 == '1' then
            return TTBR1_EL1.ASID;
        else
            return TTBR0_EL1.ASID;

    else
        if TTBCR.EAE == '0' then
            return ZeroExtend(CONTEXTIDR.ASID, 16);
        else
            if TTBCR.A1 == '1' then
                return ZeroExtend(TTBR1.ASID, 16);
            else
                return ZeroExtend(TTBR0.ASID, 16);
```

Library pseudocode for shared/functions/predictionrestrict/ExecutionCntxt

```
// ExecutionCntxt
// ===============
// Context information for prediction restriction operation.

type ExecutionCntxt is (
    boolean          is_vmid_valid, // is vmid valid for current context
    boolean          all_vmid,      // should the operation be applied for all vmids
    bits(16)         vmid,          // if all_vmid = FALSE, vmid to which operation is applied
    boolean          is_asid_valid, // is asid valid for current context
    boolean          all_asid,      // should the operation be applied for all asids
    bits(16)         asid,          // if all_asid = FALSE, ASID to which operation is applied
    bits(2)          target_el,     // target EL at which operation is performed
    SecurityState    security,
    RestrictType     restriction    // type of restriction operation
)
```

Library pseudocode for shared/functions/predictionrestrict/RESTRICT_PREDICTIONS

```
// RESTRICT_PREDICTIONS()
// ======================
// Clear all speculated values.

RESTRICT_PREDICTIONS(ExecutionCntxt c)
    IMPLEMENTATION_DEFINED;
```

Library pseudocode for shared/functions/predictionrestrict/RestrictType

```
// RestrictType
// ============
// Type of restriction on speculation.

enumeration RestrictType {
    RestrictType_DataValue,
    RestrictType_ControlFlow,
    RestrictType_CachePrefetch,
    RestrictType_Other              // Any other trained speculation mechanisms than those above
};
```

Library pseudocode for shared/functions/predictionrestrict/TargetSecurityState

```
// TargetSecurityState()
// =====================
// Decode the target security state for the prediction context.

SecurityState TargetSecurityState(bit NS, bit NSE)
    curr_ss =  SecurityStateAtEL(PSTATE.EL);
    if curr_ss == SS_NonSecure then
        return SS_NonSecure;
    elsif curr_ss == SS_Secure then
        case NS of
            when '0' return SS_Secure;
            when '1' return SS_NonSecure;
    elsif HaveRME() then
        if curr_ss == SS_Root then
            case NSE:NS of
                when '00' return SS_Secure;
                when '01' return SS_NonSecure;
                when '11' return SS_Realm;
                when '10' return SS_Root;
        elsif curr_ss == SS_Realm then
            return SS_Realm;
```

Library pseudocode for shared/functions/registers/BranchTo

```
// BranchTo()
// ==========
// Set program counter to a new address, with a branch type.
// Parameter branch_conditional indicates whether the executed branch has a conditional encoding.
// In AArch64 state the address might include a tag in the top eight bits.

BranchTo(bits(N) target, BranchType branch_type, boolean branch_conditional)
    Hint_Branch(branch_type);
    if N == 32 then
        assert UsingAArch32();
        _PC = ZeroExtend(target, 64);
    else
        assert N == 64 && !UsingAArch32();
        bits(64) target_vaddress = AArch64.BranchAddr(target<63:0>, PSTATE.EL);
        if (HaveBRBExt() &&
            branch_type IN {BranchType_DIR, BranchType_INDIR,
                            BranchType_DIRCALL, BranchType_INDCALL,
                            BranchType_RET}) then
            BRBEBranch(branch_type, branch_conditional, target_vaddress);
        boolean branch_taken = TRUE;

        if HaveStatisticalProfiling() then
            SPEBranch(target, branch_type, branch_conditional, branch_taken);

        _PC = target_vaddress;
    return;
```

Library pseudocode for shared/functions/registers/BranchToAddr

```
// BranchToAddr()
// ==============
// Set program counter to a new address, with a branch type.
// In AArch64 state the address does not include a tag in the top eight bits.

BranchToAddr(bits(N) target, BranchType branch_type)
    Hint_Branch(branch_type);
    if N == 32 then
        assert UsingAArch32();
        _PC = ZeroExtend(target, 64);
    else
        assert N == 64 && !UsingAArch32();
        _PC = target<63:0>;
    return;
```

Library pseudocode for shared/functions/registers/BranchType

```
// BranchType
// ==========
// Information associated with a change in control flow.

enumeration BranchType {
    BranchType_DIRCALL,     // Direct Branch with link
    BranchType_INDCALL,     // Indirect Branch with link
    BranchType_ERET,        // Exception return (indirect)
    BranchType_DBGEXIT,     // Exit from Debug state
    BranchType_RET,         // Indirect branch with function return hint
    BranchType_DIR,         // Direct branch
    BranchType_INDIR,       // Indirect branch
    BranchType_EXCEPTION,   // Exception entry
    BranchType_TMFAIL,      // Transaction failure
    BranchType_RESET,       // Reset
    BranchType_UNKNOWN};   // Other
```

Library pseudocode for shared/functions/registers/Hint_Branch

```
// Hint_Branch()
// =============
// Report the hint passed to BranchTo() and BranchToAddr(), for consideration when processing
// the next instruction.

Hint_Branch(BranchType hint);
```

Library pseudocode for shared/functions/registers/NextInstrAddr

```
// NextInstrAddr()
// ===============
// Return address of the sequentially next instruction.

bits(N) NextInstrAddr(integer N);
```

Library pseudocode for shared/functions/registers/ResetExternalDebugRegisters

```
// ResetExternalDebugRegisters()
// =============================
// Reset the External Debug registers in the Core power domain.

ResetExternalDebugRegisters(boolean cold_reset);
```

Library pseudocode for shared/functions/registers/ThisInstrAddr

```
// ThisInstrAddr()
// ===============
// Return address of the current instruction.

bits(N) ThisInstrAddr(integer N)
    assert N == 64 || (N == 32 && UsingAArch32());
    return _PC<N-1:0>;
```

Library pseudocode for shared/functions/registers/_PC

```
bits(64) _PC;
```

Library pseudocode for shared/functions/registers/_R

```
// _R[] - the general-purpose register file
// ========================================

array bits(64) _R[0..30];
```

Library pseudocode for shared/functions/sysregisters/SPSR

```
// SPSR[] - non-assignment form
// ============================

bits(N) SPSR[]
    bits(N) result;
    if UsingAArch32() then
        assert N == 32;
        case PSTATE.M of
            when M32_FIQ      result = SPSR_fiq<N-1:0>;
            when M32_IRQ      result = SPSR_irq<N-1:0>;
            when M32_Svc      result = SPSR_svc<N-1:0>;
            when M32_Monitor  result = SPSR_mon<N-1:0>;
            when M32_Abort    result = SPSR_abt<N-1:0>;
            when M32_Hyp      result = SPSR_hyp<N-1:0>;
            when M32_Undef    result = SPSR_und<N-1:0>;
            otherwise         Unreachable();
    else
        assert N == 64;
        case PSTATE.EL of
            when EL1          result = SPSR_EL1<N-1:0>;
            when EL2          result = SPSR_EL2<N-1:0>;
            when EL3          result = SPSR_EL3<N-1:0>;
            otherwise         Unreachable();
    return result;

// SPSR[] - assignment form
// ========================

SPSR[] = bits(N) value
    if UsingAArch32() then
        assert N == 32;
        case PSTATE.M of
            when M32_FIQ      SPSR_fiq<N-1:0> = value<N-1:0>;
            when M32_IRQ      SPSR_irq<N-1:0> = value<N-1:0>;
            when M32_Svc      SPSR_svc<N-1:0> = value<N-1:0>;
            when M32_Monitor  SPSR_mon<N-1:0> = value<N-1:0>;
            when M32_Abort    SPSR_abt<N-1:0> = value<N-1:0>;
            when M32_Hyp      SPSR_hyp<N-1:0> = value<N-1:0>;
            when M32_Undef    SPSR_und<N-1:0> = value<N-1:0>;
            otherwise         Unreachable();
    else
        assert N == 64;
        case PSTATE.EL of
            when EL1          SPSR_EL1<N-1:0> = value<N-1:0>;
            when EL2          SPSR_EL2<N-1:0> = value<N-1:0>;
            when EL3          SPSR_EL3<N-1:0> = value<N-1:0>;
            otherwise         Unreachable();
    return;
```

Library pseudocode for shared/functions/system/AArch64.ChkFeat

```
// AArch64.ChkFeat()
// =================
// Indicates the status of some features

bits(64) AArch64.ChkFeat(bits(64) feat_select)
    bits(64) feat_en = Zeros(64);
    feat_en[0] = if HaveGCS() && GCSEnabled(PSTATE.EL) then '1' else '0';
    return feat_select AND NOT(feat_en);
```

Library pseudocode for shared/functions/system/BranchTargetCheck

```
// BranchTargetCheck()
// ===================
// This function is executed checks if the current instruction is a valid target for a branch
// taken into, or inside, a guarded page. It is executed on every cycle once the current
// instruction has been decoded and the values of InGuardedPage and BTypeCompatible have been
// determined for the current instruction.

BranchTargetCheck()
    assert HaveBTIExt() && !UsingAArch32();

    // The branch target check considers two state variables:
    // * InGuardedPage, which is evaluated during instruction fetch.
    // * BTypeCompatible, which is evaluated during instruction decode.
    if InGuardedPage && PSTATE.BTYPE != '00' && !BTypeCompatible && !Halted() then
        bits(64) pc = ThisInstrAddr(64);
        AArch64.BranchTargetException(pc<51:0>);

    boolean branch_instr = AArch64.ExecutingBROrBLROrRetInstr();
    boolean bti_instr    = AArch64.ExecutingBTIInstr();

    // PSTATE.BTYPE defaults to 00 for instructions that do not explictly set BTYPE.
    if !(branch_instr || bti_instr) then
        BTypeNext = '00';
```

Library pseudocode for shared/functions/system/ClearEventRegister

```
// ClearEventRegister()
// ====================
// Clear the Event Register of this PE.

ClearEventRegister()
    EventRegister = '0';
    return;
```

Library pseudocode for shared/functions/system/ConditionHolds

```
// ConditionHolds()
// ================
// Return TRUE iff COND currently holds

boolean ConditionHolds(bits(4) cond)
    // Evaluate base condition.
    boolean result;
    case cond<3:1> of
        when '000' result = (PSTATE.Z == '1');                          // EQ or NE
        when '001' result = (PSTATE.C == '1');                          // CS or CC
        when '010' result = (PSTATE.N == '1');                          // MI or PL
        when '011' result = (PSTATE.V == '1');                          // VS or VC
        when '100' result = (PSTATE.C == '1' && PSTATE.Z == '0');       // HI or LS
        when '101' result = (PSTATE.N == PSTATE.V);                     // GE or LT
        when '110' result = (PSTATE.N == PSTATE.V && PSTATE.Z == '0');  // GT or LE
        when '111' result = TRUE;                                       // AL

    // Condition flag values in the set '111x' indicate always true
    // Otherwise, invert condition if necessary.
    if cond<0> == '1' && cond != '1111' then
        result = !result;

    return result;
```

Library pseudocode for shared/functions/system/ConsumptionOfSpeculativeDataBarrier

```
// ConsumptionOfSpeculativeDataBarrier()
// =====================================

ConsumptionOfSpeculativeDataBarrier();
```

Library pseudocode for shared/functions/system/CurrentInstrSet

```
// CurrentInstrSet()
// =================

InstrSet CurrentInstrSet()
    InstrSet result;
    if UsingAArch32() then
        result = if PSTATE.T == '0' then InstrSet_A32 else InstrSet_T32;
        // PSTATE.J is RES0. Implementation of T32EE or Jazelle state not permitted.
    else
        result = InstrSet_A64;
    return result;
```

Library pseudocode for shared/functions/system/CurrentPL

```
// CurrentPL()
// ===========

PrivilegeLevel CurrentPL()
    return PLOfEL(PSTATE.EL);
```

Library pseudocode for shared/functions/system/CurrentSecurityState

```
// CurrentSecurityState()
// ======================
// Returns the effective security state at the exception level based off current settings.

SecurityState CurrentSecurityState()
    return SecurityStateAtEL(PSTATE.EL);
```

Library pseudocode for shared/functions/system/DSBAlias

```
// DSBAlias
// ========
// Aliases of DSB.

enumeration DSBAlias {DSBAlias_SSBB, DSBAlias_PSSBB, DSBAlias_DSB};
```

Library pseudocode for shared/functions/system/EL0

```
constant bits(2) EL3 = '11';
constant bits(2) EL2 = '10';
constant bits(2) EL1 = '01';
constant bits(2) EL0 = '00';
```

Library pseudocode for shared/functions/system/EL2Enabled

```
// EL2Enabled()
// ============
// Returns TRUE if EL2 is present and executing
// - with the PE in Non-secure state when Non-secure EL2 is implemented, or
// - with the PE in Realm state when Realm EL2 is implemented, or
// - with the PE in Secure state when Secure EL2 is implemented and enabled, or
// - when EL3 is not implemented.

boolean EL2Enabled()
    return HaveEL(EL2) && (!HaveEL(EL3) || SCR_GEN[].NS == '1' || IsSecureEL2Enabled());
```

Library pseudocode for shared/functions/system/EL3SDDUndef

```
// EL3SDDUndef()
// =============
// Returns TRUE if in Debug state and EDSCR.SDD is set.

boolean EL3SDDUndef()
    return Halted() && EDSCR.SDD == '1';
```

Library pseudocode for shared/functions/system/EL3SDDUndefPriority

```
// EL3SDDUndefPriority()
// =====================
// Returns TRUE if in Debug state, EDSCR.SDD is set, and an EL3 trap by an
// EL3 control register has priority over other traps.
// The IMPLEMENTATION DEFINED priority may be different for each case.

boolean EL3SDDUndefPriority()
    return (Halted() && EDSCR.SDD == '1' &&
            boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'");
```

Library pseudocode for shared/functions/system/ELFromM32

```
// ELFromM32()
// ===========

(boolean,bits(2)) ELFromM32(bits(5) mode)
    // Convert an AArch32 mode encoding to an Exception level.
    // Returns (valid,EL):
    //   'valid' is TRUE if 'mode<4:0>' encodes a mode that is both valid for this implementation
    //           and the current value of SCR.NS/SCR_EL3.NS.
    //   'EL'    is the Exception level decoded from 'mode'.
    bits(2) el;
    boolean valid = !BadMode(mode);  // Check for modes that are not valid for this implementation
    bits(2) effective_nse_ns = EffectiveSCR_EL3_NSE() : EffectiveSCR_EL3_NS();

    case mode of
        when M32_Monitor
            el = EL3;
        when M32_Hyp
            el = EL2;
        when M32_FIQ, M32_IRQ, M32_Svc, M32_Abort, M32_Undef, M32_System
            // If EL3 is implemented and using AArch32, then these modes are EL3 modes in Secure
            // state, and EL1 modes in Non-secure state. If EL3 is not implemented or is using
            // AArch64, then these modes are EL1 modes.
            el = (if HaveEL(EL3) && !HaveAArch64() && SCR.NS == '0' then EL3 else EL1);
        when M32_User
            el = EL0;
        otherwise
            valid = FALSE;           // Passed an illegal mode value

    if valid && el == EL2 && HaveEL(EL3) && SCR_GEN[].NS == '0' then
        valid = FALSE;               // EL2 only valid in Non-secure state in AArch32

    elsif valid && HaveRME() && effective_nse_ns == '10' then
        valid = FALSE;               // Illegal Exception Return from EL3 if SCR_EL3.<NSE,NS>
                                     // selects a reserved encoding

    if !valid then el = bits(2) UNKNOWN;
    return (valid, el);
```

Library pseudocode for shared/functions/system/ELFromSPSR

```
// ELFromSPSR()
// ============

// Convert an SPSR value encoding to an Exception level.
// Returns (valid,EL):
//   'valid' is TRUE if 'spsr<4:0>' encodes a valid mode for the current state.
//   'EL'    is the Exception level decoded from 'spsr'.

(boolean,bits(2)) ELFromSPSR(bits(N) spsr)
    bits(2) el;
    boolean valid;
    bits(2) effective_nse_ns;
    if spsr<4> == '0' then      // AArch64 state
        el = spsr<3:2>;
        effective_nse_ns = EffectiveSCR_EL3_NSE() : EffectiveSCR_EL3_NS();
        if !HaveAArch64() then
            valid = FALSE;      // No AArch64 support
        elsif !HaveEL(el) then
            valid = FALSE;      // Exception level not implemented
        elsif spsr<1> == '1' then
            valid = FALSE;      // M[1] must be 0
        elsif el == EL0 && spsr<0> == '1' then
            valid = FALSE;      // for EL0, M[0] must be 0
        elsif HaveRME() && el != EL3 && effective_nse_ns == '10' then
            valid = FALSE;      // Only EL3 valid in Root state
        elsif el == EL2 && HaveEL(EL3) && !IsSecureEL2Enabled() && SCR_EL3.NS == '0' then
            valid = FALSE;      // Unless Secure EL2 is enabled, EL2 valid only in Non-secure state
        else
            valid = TRUE;
    elsif HaveAArch32() then    // AArch32 state
        (valid, el) = ELFromM32(spsr<4:0>);
    else
        valid = FALSE;

    if !valid then el = bits(2) UNKNOWN;
    return (valid,el);
```

Library pseudocode for shared/functions/system/ELIsInHost

```
// ELIsInHost()
// ============

boolean ELIsInHost(bits(2) el)
    if !HaveVirtHostExt() || ELUsingAArch32(EL2) then
        return FALSE;
    case el of
        when EL3
            return FALSE;
        when EL2
            return EL2Enabled() && HCR_EL2.E2H == '1';
        when EL1
            return FALSE;
        when EL0
            return EL2Enabled() && HCR_EL2.<E2H,TGE> == '11';
        otherwise
            Unreachable();
```

Library pseudocode for shared/functions/system/ELStateUsingAArch32

```
// ELStateUsingAArch32()
// =====================

boolean ELStateUsingAArch32(bits(2) el, boolean secure)
    // See ELStateUsingAArch32K() for description. Must only be called in circumstances where
    // result is valid (typically, that means 'el IN {EL1,EL2,EL3}').
    (known, aarch32) = ELStateUsingAArch32K(el, secure);
    assert known;
    return aarch32;
```

Library pseudocode for shared/functions/system/ELStateUsingAArch32K

```
// ELStateUsingAArch32K()
// ======================

(boolean,boolean) ELStateUsingAArch32K(bits(2) el, boolean secure)
    // Returns (known, aarch32):
    //   'known'   is FALSE for EL0 if the current Exception level is not EL0 and EL1 is
    //             using AArch64, since it cannot determine the state of EL0; TRUE otherwise.
    //   'aarch32' is TRUE if the specified Exception level is using AArch32; FALSE otherwise.
    if !HaveAArch32EL(el) then
        return (TRUE, FALSE);   // Exception level is using AArch64
    elsif secure && el == EL2 then
        return (TRUE, FALSE);   // Secure EL2 is using AArch64
    elsif !HaveAArch64() then
        return (TRUE, TRUE);    // Highest Exception level, therefore all levels are using AArch32

    // Remainder of function deals with the interprocessing cases when highest
    // Exception level is using AArch64

    boolean aarch32 = boolean UNKNOWN;
    boolean known = TRUE;

    aarch32_below_el3 = (HaveEL(EL3) && (!secure || !HaveSecureEL2Ext() || SCR_EL3.EEL2 == '0') &&
                         SCR_EL3.RW == '0');
    aarch32_at_el1 = (aarch32_below_el3 ||
                      (HaveEL(EL2) && (!secure || (HaveSecureEL2Ext() && SCR_EL3.EEL2 == '1')) &&
                       !(HaveVirtHostExt() && HCR_EL2.<E2H,TGE> == '11') &&
                       HCR_EL2.RW == '0'));
    if el == EL0 && !aarch32_at_el1 then       // Only know if EL0 using AArch32 from PSTATE
        if PSTATE.EL == EL0 then
            aarch32 = PSTATE.nRW == '1';       // EL0 controlled by PSTATE
        else
            known = FALSE;                     // EL0 state is UNKNOWN
    else
        aarch32 = (aarch32_below_el3 && el != EL3) || (aarch32_at_el1 && el IN {EL1,EL0});

    if !known then aarch32 = boolean UNKNOWN;
    return (known, aarch32);
```

Library pseudocode for shared/functions/system/ELUsingAArch32

```
// ELUsingAArch32()
// ================

boolean ELUsingAArch32(bits(2) el)
    return ELStateUsingAArch32(el, IsSecureBelowEL3());
```

Library pseudocode for shared/functions/system/ELUsingAArch32K

```
// ELUsingAArch32K()
// =================

(boolean,boolean) ELUsingAArch32K(bits(2) el)
    return ELStateUsingAArch32K(el, IsSecureBelowEL3());
```

Library pseudocode for shared/functions/system/EffectiveEA

```
// EffectiveEA()
// =============
// Returns effective SCR_EL3.EA value

bit EffectiveEA()
    if Halted() && EDSCR.SDD == '0' then
        return '0';
    else
        return if HaveAArch64() then SCR_EL3.EA else SCR.EA;
```

Library pseudocode for shared/functions/system/EffectiveSCR_EL3_NS

```
// EffectiveSCR_EL3_NS()
// =====================
// Return Effective SCR_EL3.NS value.

bit EffectiveSCR_EL3_NS()
    if !HaveSecureState() then
        return '1';
    elsif !HaveEL(EL3) then
        return '0';
    else
        return SCR_EL3.NS;
```

Library pseudocode for shared/functions/system/EffectiveSCR_EL3_NSE

```
// EffectiveSCR_EL3_NSE()
// ======================
// Return Effective SCR_EL3.NSE value.

bit EffectiveSCR_EL3_NSE()
    return if !HaveRME() then '0' else SCR_EL3.NSE;
```

Library pseudocode for shared/functions/system/EffectiveSCR_EL3_RW

```
// EffectiveSCR_EL3_RW()
// =====================
// Returns effective SCR_EL3.RW value

bit EffectiveSCR_EL3_RW()
    if !HaveAArch64() then
        return '0';
    if !HaveAArch32EL(EL2) && !HaveAArch32EL(EL1) then
        return '1';
    if HaveAArch32EL(EL1) then
        if !HaveAArch32EL(EL2) && SCR_EL3.NS == '1' then
            return '1';
        if HaveSecureEL2Ext() && SCR_EL3.EEL2 == '1' && SCR_EL3.NS == '0' then
            return '1';
    return SCR_EL3.RW;
```

Library pseudocode for shared/functions/system/EffectiveTGE

```
// EffectiveTGE()
// ==============
// Returns effective TGE value

bit EffectiveTGE()
    if EL2Enabled() then
        return if ELUsingAArch32(EL2) then HCR.TGE else HCR_EL2.TGE;
    else
        return '0';        // Effective value of TGE is zero
```

Library pseudocode for shared/functions/system/EndOfInstruction

```
// EndOfInstruction()
// ==================
// Terminate processing of the current instruction.

EndOfInstruction();
```

Library pseudocode for shared/functions/system/EnterLowPowerState

```
// EnterLowPowerState()
// ====================
// PE enters a low-power state.

EnterLowPowerState();
```

Library pseudocode for shared/functions/system/EventRegister

```
bits(1) EventRegister;
```

Library pseudocode for shared/functions/system/ExceptionalOccurrenceTargetState

```
// ExceptionalOccurrenceTargetState
// ================================
// Enumeration to represent the target state of an Exceptional Occurrence.
// The Exceptional Occurrence can be either Exception or Debug State entry.

enumeration ExceptionalOccurrenceTargetState {
    AArch32_NonDebugState,
    AArch64_NonDebugState,
    DebugState
};
```

Library pseudocode for shared/functions/system/FIQPending

```
// FIQPending()
// ============
// Returns a tuple indicating if there is any pending physical FIQ
// and if the pending FIQ has superpriority.

(boolean, boolean) FIQPending();
```

Library pseudocode for shared/functions/system/GetAccumulatedFPExceptions

```
// GetAccumulatedFPExceptions()
// ============================
// Returns FP exceptions accumulated by the PE.

bits(8) GetAccumulatedFPExceptions();
```

Library pseudocode for shared/functions/system/GetLoadStoreType

```
// GetLoadStoreType()
// ==================
// Returns the Load/Store Type. Used when a Translation fault,
// Access flag fault, or Permission fault generates a Data Abort.

bits(2) GetLoadStoreType();
```

Library pseudocode for shared/functions/system/GetPSRFromPSTATE

```
// GetPSRFromPSTATE()
// ==================
// Return a PSR value which represents the current PSTATE

bits(N) GetPSRFromPSTATE(ExceptionalOccurrenceTargetState targetELState, integer N)
    if UsingAArch32() && targetELState == AArch32_NonDebugState then
        assert N == 32;
    else
        assert N == 64;

    bits(N) spsr = Zeros(N);
    spsr<31:28> = PSTATE.<N,Z,C,V>;
    if HavePANExt() then spsr<22> = PSTATE.PAN;
    spsr<20>    = PSTATE.IL;
    if PSTATE.nRW == '1' then                          // AArch32 state
        spsr<27>    = PSTATE.Q;
        spsr<26:25> = PSTATE.IT<1:0>;
        if HaveSSBSExt() then spsr<23> = PSTATE.SSBS;
        if HaveDITExt() then
            if targetELState == AArch32_NonDebugState then
                spsr<21> = PSTATE.DIT;
            else                                       // AArch64_NonDebugState or DebugState
                spsr<24> = PSTATE.DIT;
        if targetELState IN {AArch64_NonDebugState, DebugState} then
            spsr<21> = PSTATE.SS;
        spsr<19:16> = PSTATE.GE;
        spsr<15:10> = PSTATE.IT<7:2>;
        spsr<9>     = PSTATE.E;
        spsr<8:6>   = PSTATE.<A,I,F>;                  // No PSTATE.D in AArch32 state
        spsr<5>     = PSTATE.T;
        assert PSTATE.M<4> == PSTATE.nRW;              // bit [4] is the discriminator
        spsr<4:0>   = PSTATE.M;
    else                                               // AArch64 state
        if HaveMTEExt() then spsr<25> = PSTATE.TCO;
        if HaveGCS() then spsr<34> = PSTATE.EXLOCK;
        if HaveDITExt() then spsr<24> = PSTATE.DIT;
        if HaveUAOExt() then spsr<23> = PSTATE.UAO;
        spsr<21>    = PSTATE.SS;
        if HaveFeatNMI() then spsr<13> = PSTATE.ALLINT;
        if HaveSSBSExt() then spsr<12> = PSTATE.SSBS;
        if HaveBTIExt() then spsr<11:10> = PSTATE.BTYPE;
        spsr<9:6>   = PSTATE.<D,A,I,F>;
        spsr<4>     = PSTATE.nRW;
        spsr<3:2>   = PSTATE.EL;
        spsr<0>     = PSTATE.SP;
    return spsr;
```

Library pseudocode for shared/functions/system/HasArchVersion

```
// HasArchVersion()
// ================
// Returns TRUE if the implemented architecture includes the extensions defined in the specified
// architecture version.

boolean HasArchVersion(ArchVersion version)
    return Variant(version);
```

Library pseudocode for shared/functions/system/HaveAArch32

```
// HaveAArch32()
// =============
// Return TRUE if AArch32 state is supported at at least EL0.

boolean HaveAArch32()
    return IsFeatureImplemented(FEAT_AA32EL0);
```

Library pseudocode for shared/functions/system/HaveAArch32EL

```
// HaveAArch32EL()
// ===============
// Return TRUE if Exception level 'el' supports AArch32 in this implementation

boolean HaveAArch32EL(bits(2) el)
    case el of
        when EL0 return IsFeatureImplemented(FEAT_AA32EL0);
        when EL1 return IsFeatureImplemented(FEAT_AA32EL1);
        when EL2 return IsFeatureImplemented(FEAT_AA32EL2);
        when EL3 return IsFeatureImplemented(FEAT_AA32EL3);
```

Library pseudocode for shared/functions/system/HaveAArch64

```
// HaveAArch64()
// =============
// Return TRUE if the highest Exception level is using AArch64 state.

boolean HaveAArch64()
    return (IsFeatureImplemented(FEAT_AA64EL0) || IsFeatureImplemented(FEAT_AA64EL1) ||
            IsFeatureImplemented(FEAT_AA64EL2) || IsFeatureImplemented(FEAT_AA64EL3));
```

Library pseudocode for shared/functions/system/HaveEL

```
// HaveEL()
// ========
// Return TRUE if Exception level 'el' is supported

boolean HaveEL(bits(2) el)
    case el of
        when EL1,EL0
            return TRUE;                         // EL1 and EL0 must exist
        when EL2
            return IsFeatureImplemented(FEAT_AA64EL2) || IsFeatureImplemented(FEAT_AA32EL2);
        when EL3
            return IsFeatureImplemented(FEAT_AA64EL3) || IsFeatureImplemented(FEAT_AA32EL3);
        otherwise
            Unreachable();
```

Library pseudocode for shared/functions/system/HaveELUsingSecurityState

```
// HaveELUsingSecurityState()
// ==========================
// Returns TRUE if Exception level 'el' with Security state 'secure' is supported,
// FALSE otherwise.

boolean HaveELUsingSecurityState(bits(2) el, boolean secure)

    case el of
        when EL3
            assert secure;
            return HaveEL(EL3);
        when EL2
            if secure then
                return HaveEL(EL2) && HaveSecureEL2Ext();
            else
                return HaveEL(EL2);
        otherwise
            return (HaveEL(EL3) ||
                    (secure == boolean IMPLEMENTATION_DEFINED "Secure-only implementation"));
```

Library pseudocode for shared/functions/system/HaveFP16Ext

```
// HaveFP16Ext()
// =============
// Return TRUE if FP16 extension is supported

boolean HaveFP16Ext()
    return IsFeatureImplemented(FEAT_FP16);
```

Library pseudocode for shared/functions/system/HaveSecureState

```
// HaveSecureState()
// =================
// Return TRUE if Secure State is supported.

boolean HaveSecureState()
    if !HaveEL(EL3) then
        return SecureOnlyImplementation();
    if HaveRME() && !HaveSecureEL2Ext() then
        return FALSE;
    return TRUE;
```

Library pseudocode for shared/functions/system/HighestEL

```
// HighestEL()
// ===========
// Returns the highest implemented Exception level.

bits(2) HighestEL()
    if HaveEL(EL3) then
        return EL3;
    elsif HaveEL(EL2) then
        return EL2;
    else
        return EL1;
```

Library pseudocode for shared/functions/system/Hint_CLRBHB

```
// Hint_CLRBHB()
// =============
// Provides a hint to clear the branch history for the current context.

Hint_CLRBHB();
```

Library pseudocode for shared/functions/system/Hint_DGH

```
// Hint_DGH()
// ==========
// Provides a hint to close any gathering occurring within the micro-architecture.

Hint_DGH();
```

Library pseudocode for shared/functions/system/Hint_WFE

```
// Hint_WFE()
// ==========
// Provides a hint indicating that the PE can enter a low-power state
// and remain there until a wakeup event occurs or, for WFET,  a local
// timeout event is generated when the virtual timer value equals or
// exceeds the supplied threshold value.

Hint_WFE(integer localtimeout, WFxType wfxtype)
    if IsEventRegisterSet() then
        ClearEventRegister();
    elsif HaveFeatWFxT() && LocalTimeoutEvent(localtimeout) then
        // No further operation if the local timeout has expired.
        EndOfInstruction();
    else
        bits(2) target_el;
        trap = FALSE;
        if PSTATE.EL == EL0 then
            // Check for traps described by the OS which may be EL1 or EL2.
            if HaveTWEDExt() then
                sctlr     = SCTLR[];
                trap      = sctlr.nTWE == '0';
                target_el = EL1;
            else
                AArch64.CheckForWFxTrap(EL1, wfxtype);
        if !trap && PSTATE.EL IN {EL0, EL1} && EL2Enabled() && !IsInHost() then
            // Check for traps described by the Hypervisor.
            if HaveTWEDExt() then
                trap      = HCR_EL2.TWE == '1';
                target_el = EL2;
            else
                AArch64.CheckForWFxTrap(EL2, wfxtype);

        if !trap && HaveEL(EL3) && PSTATE.EL != EL3 then
            // Check for traps described by the Secure Monitor.
            if HaveTWEDExt() then
                trap      = SCR_EL3.TWE == '1';
                target_el = EL3;
            else
                AArch64.CheckForWFxTrap(EL3, wfxtype);

        if trap && PSTATE.EL != EL3 then
            // Determine if trap delay is enabled and delay amount
            (delay_enabled, delay) = WFETrapDelay(target_el);
            if !WaitForEventUntilDelay(delay_enabled, delay) then
                // Event did not arrive before delay expired so trap WFE
                AArch64.WFxTrap(wfxtype, target_el);
        else
            WaitForEvent(localtimeout);
```

Library pseudocode for shared/functions/system/Hint_WFI

```
// Hint_WFI()
// ==========
// Provides a hint indicating that the PE can enter a low-power state and
// remain there until a wakeup event occurs or, for WFIT, a local timeout
// event is generated when the virtual timer value equals or exceeds the
// supplied threshold value.

Hint_WFI(integer localtimeout, WFxType wfxtype)
    if HaveTME() && TSTATE.depth > 0 then
        FailTransaction(TMFailure_ERR, FALSE);

    if InterruptPending() || (HaveFeatWFxT() && LocalTimeoutEvent(localtimeout)) then
        // No further operation if an interrupt is pending or the local timeout has expired.
        EndOfInstruction();
    else
        if PSTATE.EL == EL0 then
            // Check for traps described by the OS.
            AArch64.CheckForWFxTrap(EL1, wfxtype);
        if PSTATE.EL IN {EL0, EL1} && EL2Enabled() && !IsInHost() then
            // Check for traps described by the Hypervisor.
            AArch64.CheckForWFxTrap(EL2, wfxtype);
        if HaveEL(EL3) && PSTATE.EL != EL3 then
            // Check for traps described by the Secure Monitor.
            AArch64.CheckForWFxTrap(EL3, wfxtype);
        WaitForInterrupt(localtimeout);
```

Library pseudocode for shared/functions/system/Hint_Yield

```
// Hint_Yield()
// ============
// Provides a hint that the task performed by a thread is of low
// importance so that it could yield to improve overall performance.

Hint_Yield();
```

Library pseudocode for shared/functions/system/IRQPending

```
// IRQPending()
// ============
// Returns a tuple indicating if there is any pending physical IRQ
// and if the pending IRQ has superpriority.

(boolean, boolean) IRQPending();
```

Library pseudocode for shared/functions/system/IllegalExceptionReturn

```
// IllegalExceptionReturn()
// ========================

boolean IllegalExceptionReturn(bits(N) spsr)

    // Check for illegal return:
    //   * To an unimplemented Exception level.
    //   * To EL2 in Secure state, when SecureEL2 is not enabled.
    //   * To EL0 using AArch64 state, with SPSR.M[0]==1.
    //   * To AArch64 state with SPSR.M[1]==1.
    //   * To AArch32 state with an illegal value of SPSR.M.
    (valid, target) = ELFromSPSR(spsr);
    if !valid then return TRUE;

    // Check for return to higher Exception level
    if UInt(target) > UInt(PSTATE.EL) then return TRUE;

    spsr_mode_is_aarch32 = (spsr<4> == '1');

    // Check for illegal return:
    //   * To EL1, EL2 or EL3 with register width specified in the SPSR different from the
    //     Execution state used in the Exception level being returned to, as determined by
    //     the SCR_EL3.RW or HCR_EL2.RW bits, or as configured from reset.
    //   * To EL0 using AArch64 state when EL1 is using AArch32 state as determined by the
    //     SCR_EL3.RW or HCR_EL2.RW bits or as configured from reset.
    //   * To AArch64 state from AArch32 state (should be caught by above)
    (known, target_el_is_aarch32) = ELUsingAArch32K(target);
    assert known || (target == EL0 && !ELUsingAArch32(EL1));
    if known && spsr_mode_is_aarch32 != target_el_is_aarch32 then return TRUE;

    // Check for illegal return from AArch32 to AArch64
    if UsingAArch32() && !spsr_mode_is_aarch32 then return TRUE;

    // Check for illegal return to EL1 when HCR.TGE is set and when either of
    // * SecureEL2 is enabled.
    // * SecureEL2 is not enabled and EL1 is in Non-secure state.
    if HaveEL(EL2) && target == EL1 && HCR_EL2.TGE == '1' then
        if (!IsSecureBelowEL3() || IsSecureEL2Enabled()) then return TRUE;

    if (HaveGCS() && PSTATE.EXLOCK == '0' && PSTATE.EL == target &&
        GetCurrentEXLOCKEN() && !Halted()) then
        return TRUE;

    return FALSE;
```

Library pseudocode for shared/functions/system/InstrSet

```
// InstrSet
// ========

enumeration InstrSet {InstrSet_A64, InstrSet_A32, InstrSet_T32};
```

Library pseudocode for shared/functions/system/InstructionSynchronizationBarrier

```
// InstructionSynchronizationBarrier()
// ===================================
InstructionSynchronizationBarrier();
```

Library pseudocode for shared/functions/system/InterruptPending

```
// InterruptPending()
// ==================
// Returns TRUE if there are any pending physical or virtual
// interrupts, and FALSE otherwise.

boolean InterruptPending()
    boolean pending_virtual_interrupt = FALSE;
    (irq_pending, -) = IRQPending();
    (fiq_pending, -) = FIQPending();
    boolean pending_physical_interrupt = (irq_pending || fiq_pending ||
                                          IsPhysicalSErrorPending());

    if EL2Enabled() && PSTATE.EL IN {EL0, EL1} && HCR_EL2.TGE == '0' then
        boolean virq_pending = HCR_EL2.IMO == '1' && (VirtualIRQPending() || HCR_EL2.VI == '1') ;
        boolean vfiq_pending = HCR_EL2.FMO == '1' && (VirtualFIQPending() || HCR_EL2.VF == '1');
        boolean vsei_pending = HCR_EL2.AMO == '1' && (IsVirtualSErrorPending() ||
                                                      HCR_EL2.VSE == '1');
        pending_virtual_interrupt = vsei_pending || virq_pending || vfiq_pending;

    return pending_physical_interrupt || pending_virtual_interrupt;
```

Library pseudocode for shared/functions/system/IsASEInstruction

```
// IsASEInstruction()
// ==================
// Returns TRUE if the current instruction is an ASIMD or SVE vector instruction.

boolean IsASEInstruction();
```

Library pseudocode for shared/functions/system/IsCMOWControlledInstruction

```
// IsCMOWControlledInstruction()
// =============================
// When using AArch64, returns TRUE if the current instruction is one of IC IVAU,
// DC CIVAC, DC CIGDVAC, or DC CIGVAC.
// When using AArch32, returns TRUE if the current instruction is ICIMVAU or DCCIMVAC.

boolean IsCMOWControlledInstruction();
```

Library pseudocode for shared/functions/system/IsCurrentSecurityState

```
// IsCurrentSecurityState()
// ========================
// Returns TRUE if the current Security state matches
// the given Security state, and FALSE otherwise.

boolean IsCurrentSecurityState(SecurityState ss)
    return CurrentSecurityState() == ss;
```

Library pseudocode for shared/functions/system/IsEventRegisterSet

```
// IsEventRegisterSet()
// ====================
// Return TRUE if the Event Register of this PE is set, and FALSE if it is clear.

boolean IsEventRegisterSet()
    return EventRegister == '1';
```

Library pseudocode for shared/functions/system/IsHighestEL

```
// IsHighestEL()
// =============
// Returns TRUE if given exception level is the highest exception level implemented

boolean IsHighestEL(bits(2) el)
    return HighestEL() == el;
```

Library pseudocode for shared/functions/system/IsInHost

```
// IsInHost()
// ==========

boolean IsInHost()
    return ELIsInHost(PSTATE.EL);
```

Library pseudocode for shared/functions/system/IsSecure

```
// IsSecure()
// ==========
// Returns TRUE if current Exception level is in Secure state.

boolean IsSecure()
    if HaveEL(EL3) && !UsingAArch32() && PSTATE.EL == EL3 then
        return TRUE;
    elsif HaveEL(EL3) && UsingAArch32() && PSTATE.M == M32_Monitor then
        return TRUE;
    return IsSecureBelowEL3();
```

Library pseudocode for shared/functions/system/IsSecureBelowEL3

```
// IsSecureBelowEL3()
// ==================
// Return TRUE if an Exception level below EL3 is in Secure state
// or would be following an exception return to that level.
//
// Differs from IsSecure in that it ignores the current EL or Mode
// in considering security state.
// That is, if at AArch64 EL3 or in AArch32 Monitor mode, whether an
// exception return would pass to Secure or Non-secure state.

boolean IsSecureBelowEL3()
    if HaveEL(EL3) then
        return SCR_GEN[].NS == '0';
    elsif HaveEL(EL2) && (!HaveSecureEL2Ext() || !HaveAArch64()) then
        // If Secure EL2 is not an architecture option then we must be Non-secure.
        return FALSE;
    else
        // TRUE if processor is Secure or FALSE if Non-secure.
        return boolean IMPLEMENTATION_DEFINED "Secure-only implementation";
```

Library pseudocode for shared/functions/system/IsSecureEL2Enabled

```
// IsSecureEL2Enabled()
// ====================
// Returns TRUE if Secure EL2 is enabled, FALSE otherwise.

boolean IsSecureEL2Enabled()
    if HaveEL(EL2) && HaveSecureEL2Ext() then
        if HaveEL(EL3) then
            if !ELUsingAArch32(EL3) && SCR_EL3.EEL2 == '1' then
                return TRUE;
            else
                return FALSE;
        else
            return SecureOnlyImplementation();
    else
        return FALSE;
```

Library pseudocode for shared/functions/system/LocalTimeoutEvent

```
// LocalTimeoutEvent()
// ===================
// Returns TRUE if CNTVCT_EL0 equals or exceeds the localtimeout value.

boolean LocalTimeoutEvent(integer localtimeout);
```

Library pseudocode for shared/functions/system/Mode_Bits

```
constant bits(5) M32_User    = '10000';
constant bits(5) M32_FIQ     = '10001';
constant bits(5) M32_IRQ     = '10010';
constant bits(5) M32_Svc     = '10011';
constant bits(5) M32_Monitor = '10110';
constant bits(5) M32_Abort   = '10111';
constant bits(5) M32_Hyp     = '11010';
constant bits(5) M32_Undef   = '11011';
constant bits(5) M32_System  = '11111';
```

Library pseudocode for shared/functions/system/NonSecureOnlyImplementation

```
// NonSecureOnlyImplementation()
// =============================
// Returns TRUE if the security state is always Non-secure for this implementation.

boolean NonSecureOnlyImplementation()
    return boolean IMPLEMENTATION_DEFINED "Non-secure only implementation";
```

Library pseudocode for shared/functions/system/PLOfEL

```
// PLOfEL()
// ========

PrivilegeLevel PLOfEL(bits(2) el)
    case el of
        when EL3  return if !HaveAArch64() then PL1 else PL3;
        when EL2  return PL2;
        when EL1  return PL1;
        when EL0  return PL0;
```

Library pseudocode for shared/functions/system/PSTATE

```
ProcState PSTATE;
```

Library pseudocode for shared/functions/system/PhysicalCountInt

```
// PhysicalCountInt()
// ==================
// Returns the integral part of physical count value of the System counter.

bits(64) PhysicalCountInt()
    return PhysicalCount<87:24>;
```

Library pseudocode for shared/functions/system/PrivilegeLevel

```
// PrivilegeLevel
// ==============
// Privilege Level abstraction.

enumeration PrivilegeLevel {PL3, PL2, PL1, PL0};
```

Library pseudocode for shared/functions/system/ProcState

```
// ProcState
// =========
// Armv8 processor state bits.
// There is no significance to the field order.

type ProcState is (
    bits (1) N,        // Negative condition flag
    bits (1) Z,        // Zero condition flag
    bits (1) C,        // Carry condition flag
    bits (1) V,        // Overflow condition flag
    bits (1) D,        // Debug mask bit                     [AArch64 only]
    bits (1) A,        // SError interrupt mask bit
    bits (1) I,        // IRQ mask bit
    bits (1) F,        // FIQ mask bit
    bits (1) EXLOCK,   // Lock exception return state
    bits (1) PAN,      // Privileged Access Never Bit        [v8.1]
    bits (1) UAO,      // User Access Override               [v8.2]
    bits (1) DIT,      // Data Independent Timing            [v8.4]
    bits (1) TCO,      // Tag Check Override                 [v8.5, AArch64 only]
    bits (1) PM,       // PMU exception Mask
    bits (1) PPEND,     // synchronous PMU exception to be observed
    bits (2) BTYPE,    // Branch Type                        [v8.5]
    bits (1) ZA,       // Accumulation array enabled         [SME]
    bits (1) SM,       // Streaming SVE mode enabled         [SME]
    bits (1) ALLINT,   // Interrupt mask bit
    bits (1) SS,       // Software step bit
    bits (1) IL,       // Illegal Execution state bit
    bits (2) EL,       // Exception level
    bits (1) nRW,      // Execution state: 0=AArch64, 1=AArch32
    bits (1) SP,       // Stack pointer select: 0=SP0, 1=SPx [AArch64 only]
    bits (1) Q,        // Cumulative saturation flag         [AArch32 only]
    bits (4) GE,       // Greater than or Equal flags        [AArch32 only]
    bits (1) SSBS,     // Speculative Store Bypass Safe
    bits (8) IT,       // If-then bits, RES0 in CPSR         [AArch32 only]
    bits (1) J,        // J bit, RES0                        [AArch32 only, RES0 in SPSR and CPSR]
    bits (1) T,        // T32 bit, RES0 in CPSR              [AArch32 only]
    bits (1) E,        // Endianness bit                     [AArch32 only]
    bits (5) M         // Mode field                         [AArch32 only]
)
```

Library pseudocode for shared/functions/system/RestoredITBits

```
// RestoredITBits()
// ================
// Get the value of PSTATE.IT to be restored on this exception return.

bits(8) RestoredITBits(bits(N) spsr)
    it = spsr<15:10,26:25>;

    // When PSTATE.IL is set, it is CONSTRAINED UNPREDICTABLE whether the IT bits are each set
    // to zero or copied from the SPSR.
    if PSTATE.IL == '1' then
        if ConstrainUnpredictableBool(Unpredictable_ILZEROIT) then return '00000000';
        else return it;

    // The IT bits are forced to zero when they are set to a reserved value.
    if !IsZero(it<7:4>) && IsZero(it<3:0>) then
        return '00000000';

    // The IT bits are forced to zero when returning to A32 state, or when returning to an EL
    // with the ITD bit set to 1, and the IT bits are describing a multi-instruction block.
    itd = if PSTATE.EL == EL2 then HSCTLR.ITD else SCTLR.ITD;
    if (spsr<5> == '0' && !IsZero(it)) || (itd == '1' && !IsZero(it<2:0>)) then
        return '00000000';
    else
        return it;
```

Library pseudocode for shared/functions/system/SCRType

```
type SCRType;
```

Library pseudocode for shared/functions/system/SCR_GEN

```
// SCR_GEN[]
// =========

SCRType SCR_GEN[]
    // AArch32 secure & AArch64 EL3 registers are not architecturally mapped
    assert HaveEL(EL3);
    bits(64) r;
    if !HaveAArch64() then
        r = ZeroExtend(SCR, 64);
    else
        r = SCR_EL3;
    return r;
```

Library pseudocode for shared/functions/system/SecureOnlyImplementation

```
// SecureOnlyImplementation()
// ==========================
// Returns TRUE if the security state is always Secure for this implementation.

boolean SecureOnlyImplementation()
    return boolean IMPLEMENTATION_DEFINED "Secure-only implementation";
```

Library pseudocode for shared/functions/system/SecurityState

```
// SecurityState
// =============
// The Security state of an execution context

enumeration SecurityState {
    SS_NonSecure,
    SS_Root,
    SS_Realm,
    SS_Secure
};
```

Library pseudocode for shared/functions/system/SecurityStateAtEL

```
// SecurityStateAtEL()
// ===================
// Returns the effective security state at the exception level based off current settings.

SecurityState SecurityStateAtEL(bits(2) EL)
    if HaveRME() then
        if EL == EL3 then return SS_Root;
        effective_nse_ns = SCR_EL3.NSE : EffectiveSCR_EL3_NS();
        case effective_nse_ns of
            when '00' if HaveSecureEL2Ext() then return SS_Secure; else Unreachable();
            when '01' return SS_NonSecure;
            when '11' return SS_Realm;
            otherwise        Unreachable();

    if !HaveEL(EL3) then
        if SecureOnlyImplementation() then
            return SS_Secure;
        else
            return SS_NonSecure;
    elsif EL == EL3 then
        return SS_Secure;
    else
        // For EL2 call only when EL2 is enabled in current security state
        assert(EL != EL2 || EL2Enabled());
        if !ELUsingAArch32(EL3) then
            return if SCR_EL3.NS == '1' then SS_NonSecure else SS_Secure;
        else
            return if SCR.NS == '1' then SS_NonSecure else SS_Secure;
```

Library pseudocode for shared/functions/system/SendEvent

```
// SendEvent()
// ===========
// Signal an event to all PEs in a multiprocessor system to set their Event Registers.
// When a PE executes the SEV instruction, it causes this function to be executed.

SendEvent();
```

Library pseudocode for shared/functions/system/SendEventLocal

```
// SendEventLocal()
// ================
// Set the local Event Register of this PE.
// When a PE executes the SEVL instruction, it causes this function to be executed.

SendEventLocal()
    EventRegister = '1';
    return;
```

Library pseudocode for shared/functions/system/SetAccumulatedFPExceptions

```
// SetAccumulatedFPExceptions()
// ============================
// Stores FP Exceptions accumulated by the PE.

SetAccumulatedFPExceptions(bits(8) accumulated_exceptions);
```

Library pseudocode for shared/functions/system/SetPSTATEFromPSR

```
// SetPSTATEFromPSR()
// ==================

SetPSTATEFromPSR(bits(N) spsr)
    boolean illegal_psr_state = IllegalExceptionReturn(spsr);
    SetPSTATEFromPSR(spsr, illegal_psr_state);

// SetPSTATEFromPSR()
// ==================
// Set PSTATE based on a PSR value

SetPSTATEFromPSR(bits(N) spsr_in, boolean illegal_psr_state)
    bits(N) spsr = spsr_in;
    boolean from_aarch64 = !UsingAArch32();
    PSTATE.SS = DebugExceptionReturnSS(spsr);

    ShouldAdvanceSS = FALSE;
    if illegal_psr_state then
        PSTATE.IL = '1';
        if HaveSSBSExt() then PSTATE.SSBS = bit UNKNOWN;
        if HaveBTIExt() then PSTATE.BTYPE = bits(2) UNKNOWN;
        if HaveUAOExt() then PSTATE.UAO = bit UNKNOWN;
        if HaveDITExt() then PSTATE.DIT = bit UNKNOWN;
        if HaveMTEExt() then PSTATE.TCO = bit UNKNOWN;
    else
        // State that is reinstated only on a legal exception return
        PSTATE.IL = spsr<20>;
        if spsr<4> == '1' then                    // AArch32 state
            AArch32.WriteMode(spsr<4:0>);         // Sets PSTATE.EL correctly
            if HaveSSBSExt() then PSTATE.SSBS = spsr<23>;
        else                                      // AArch64 state
            PSTATE.nRW = '0';
            PSTATE.EL  = spsr<3:2>;
            PSTATE.SP  = spsr<0>;
            if HaveBTIExt() then PSTATE.BTYPE = spsr<11:10>;
            if HaveSSBSExt() then PSTATE.SSBS = spsr<12>;
            if HaveUAOExt() then PSTATE.UAO = spsr<23>;
            if HaveDITExt() then PSTATE.DIT = spsr<24>;
            if HaveMTEExt() then PSTATE.TCO = spsr<25>;
            if HaveGCS() then PSTATE.EXLOCK = spsr<34>;

    // If PSTATE.IL is set, it is CONSTRAINED UNPREDICTABLE whether the T bit is set to zero or
    // copied from SPSR.
    if PSTATE.IL == '1' && PSTATE.nRW == '1' then
        if ConstrainUnpredictableBool(Unpredictable_ILZEROT) then spsr<5> = '0';

    // State that is reinstated regardless of illegal exception return
    PSTATE.<N,Z,C,V> = spsr<31:28>;
    if HavePANExt() then PSTATE.PAN = spsr<22>;
    if PSTATE.nRW == '1' then                     // AArch32 state
        PSTATE.Q         = spsr<27>;
        PSTATE.IT        = RestoredITBits(spsr);
        ShouldAdvanceIT  = FALSE;
        if HaveDITExt() then
            PSTATE.DIT = (if (Restarting() || from_aarch64) then spsr<24> else spsr<21>);
        PSTATE.GE        = spsr<19:16>;
        PSTATE.E         = spsr<9>;
        PSTATE.<A,I,F>   = spsr<8:6>;             // No PSTATE.D in AArch32 state
        PSTATE.T         = spsr<5>;               // PSTATE.J is RES0
    else                                          // AArch64 state
        if HaveFeatNMI() then PSTATE.ALLINT = spsr<13>;
        PSTATE.<D,A,I,F> = spsr<9:6>;             // No PSTATE.<Q,IT,GE,E,T> in AArch64 state
    return;
```

Library pseudocode for shared/functions/system/ShouldAdvanceIT

```
boolean ShouldAdvanceIT;
```

Library pseudocode for shared/functions/system/ShouldAdvanceSS

```
boolean ShouldAdvanceSS;
```

Library pseudocode for shared/functions/system/SpeculationBarrier

```
// SpeculationBarrier()
// ====================

SpeculationBarrier();
```

Library pseudocode for shared/functions/system/SynchronizeContext

```
// SynchronizeContext()
// ====================

SynchronizeContext();
```

Library pseudocode for shared/functions/system/SynchronizeErrors

```
// SynchronizeErrors()
// ===================
// Implements the error synchronization event.

SynchronizeErrors();
```

Library pseudocode for shared/functions/system/TakeUnmaskedPhysicalSErrorInterrupts

```
// TakeUnmaskedPhysicalSErrorInterrupts()
// ======================================
// Take any pending unmasked physical SError interrupt.

TakeUnmaskedPhysicalSErrorInterrupts(boolean iesb_req);
```

Library pseudocode for shared/functions/system/TakeUnmaskedSErrorInterrupts

```
// TakeUnmaskedSErrorInterrupts()
// ==============================
// Take any pending unmasked physical SError interrupt or unmasked virtual SError
// interrupt.

TakeUnmaskedSErrorInterrupts();
```

Library pseudocode for shared/functions/system/ThisInstr

```
// ThisInstr()
// ===========

bits(32) ThisInstr();
```

Library pseudocode for shared/functions/system/ThisInstrLength

```
// ThisInstrLength()
// =================

integer ThisInstrLength();
```

Library pseudocode for shared/functions/system/Unreachable

```
// Unreachable()
// =============

Unreachable()
    assert FALSE;
```

Library pseudocode for shared/functions/system/UsingAArch32

```
// UsingAArch32()
// ==============
// Return TRUE if the current Exception level is using AArch32, FALSE if using AArch64.

boolean UsingAArch32()
    boolean aarch32 = (PSTATE.nRW == '1');
    if !HaveAArch32() then assert !aarch32;
    if !HaveAArch64() then assert aarch32;
    return aarch32;
```

Library pseudocode for shared/functions/system/ValidSecurityStateAtEL

```
// ValidSecurityStateAtEL()
// ========================
// Returns TRUE if the current settings and architecture choices for this
// implementation permit a valid Security state at the indicated EL.

boolean ValidSecurityStateAtEL(bits(2) el)
    if !HaveEL(el) then
        return FALSE;

    if el == EL3 then
        return TRUE;

    if HaveRME() then
        bits(2) effective_nse_ns = SCR_EL3.NSE : EffectiveSCR_EL3_NS();
        if effective_nse_ns == '10' then
            return FALSE;

    if el == EL2 then
        return EL2Enabled();

    return TRUE;
```

Library pseudocode for shared/functions/system/VirtualFIQPending

```
// VirtualFIQPending()
// ===================
// Returns TRUE if there is any pending virtual FIQ.

boolean VirtualFIQPending();
```

Library pseudocode for shared/functions/system/VirtualIRQPending

```
// VirtualIRQPending()
// ===================
// Returns TRUE if there is any pending virtual IRQ.

boolean VirtualIRQPending();
```

Library pseudocode for shared/functions/system/WFxType

```
// WFxType
// =======
// WFx instruction types.

enumeration WFxType {WFxType_WFE, WFxType_WFI, WFxType_WFET, WFxType_WFIT};
```

Library pseudocode for shared/functions/system/WaitForEvent

```
// WaitForEvent()
// ==============
// PE optionally suspends execution until one of the following occurs:
// - A WFE wakeup event.
// - A reset.
// - The implementation chooses to resume execution.
// - A Wait for Event with Timeout (WFET) is executing, and a local timeout event occurs
// It is IMPLEMENTATION DEFINED whether restarting execution after the period of
// suspension causes the Event Register to be cleared.

WaitForEvent(integer localtimeout)
    if !(IsEventRegisterSet() || (HaveFeatWFxT() && LocalTimeoutEvent(localtimeout))) then
        EnterLowPowerState();
    return;
```

Library pseudocode for shared/functions/system/WaitForInterrupt

```
// WaitForInterrupt()
// ==================
// PE optionally suspends execution until one of the following occurs:
// - A WFI wakeup event.
// - A reset.
// - The implementation chooses to resume execution.
// - A Wait for Interrupt with Timeout (WFIT) is executing, and a local timeout event occurs.

WaitForInterrupt(integer localtimeout)
    if !(HaveFeatWFxT() && LocalTimeoutEvent(localtimeout)) then
        EnterLowPowerState();
    return;
```

Library pseudocode for shared/functions/unpredictable/ConstrainUnpredictable

```
// ConstrainUnpredictable()
// ========================
// Return the appropriate Constraint result to control the caller's behavior.
// The return value is IMPLEMENTATION DEFINED within a permitted list for each
// UNPREDICTABLE case.
// (The permitted list is determined by an assert or case statement at the call site.)

Constraint ConstrainUnpredictable(Unpredictable which);
```

Library pseudocode for shared/functions/unpredictable/ConstrainUnpredictableBits

```
// ConstrainUnpredictableBits()
// ============================

// This is a variant of ConstrainUnpredictable for when the result can be Constraint_UNKNOWN.
// If the result is Constraint_UNKNOWN then the function also returns UNKNOWN value, but that
// value is always an allocated value; that is, one for which the behavior is not itself
// CONSTRAINED.

(Constraint,bits(width)) ConstrainUnpredictableBits(Unpredictable which, integer width);
```

Library pseudocode for shared/functions/unpredictable/ConstrainUnpredictableBool

```
// ConstrainUnpredictableBool()
// ============================
// This is a variant of the ConstrainUnpredictable function where the result is either
// Constraint_TRUE or Constraint_FALSE.

boolean ConstrainUnpredictableBool(Unpredictable which);
```

Library pseudocode for shared/functions/unpredictable/ConstrainUnpredictableInteger

```
// ConstrainUnpredictableInteger()
// ===============================
// This is a variant of ConstrainUnpredictable for when the result can be Constraint_UNKNOWN.
// If the result is Constraint_UNKNOWN then the function also returns an UNKNOWN
// value in the range low to high, inclusive.

(Constraint,integer) ConstrainUnpredictableInteger(integer low, integer high,
                                                   Unpredictable which);
```

Library pseudocode for shared/functions/unpredictable/ConstrainUnpredictableProcedure

```
// ConstrainUnpredictableProcedure()
// =================================
// This is a variant of ConstrainUnpredictable that implements a Constrained
// Unpredictable behavior for a given Unpredictable situation.
// The behavior is within permitted behaviors for a given Unpredictable situation,
// these are documented in the textual part of the architecture specification.
//
// This function is expected to be refined in an IMPLEMENTATION DEFINED manner.
// The details of possible outcomes may not be present in the code and must be interpreted
// for each use with respect to the CONSTRAINED UNPREDICTABLE specifications
// for the specific area.

ConstrainUnpredictableProcedure(Unpredictable which);
```

Library pseudocode for shared/functions/unpredictable/Constraint

```
// Constraint
// ==========
// List of Constrained Unpredictable behaviors.

enumeration Constraint    {// General
                           Constraint_NONE,              // Instruction executes with
                                                         // no change or side-effect
                                                         // to its described behavior
                           Constraint_UNKNOWN,           // Destination register
                                                         // has UNKNOWN value
                           Constraint_UNDEF,             // Instruction is UNDEFINED
                           Constraint_UNDEFEL0,          // Instruction is UNDEFINED at EL0 only
                           Constraint_NOP,               // Instruction executes as NOP
                           Constraint_TRUE,
                           Constraint_FALSE,
                           Constraint_DISABLED,
                           Constraint_UNCOND,            // Instruction executes unconditionally
                           Constraint_COND,              // Instruction executes conditionally
                           Constraint_ADDITIONAL_DECODE, // Instruction executes
                                                         // with additional decode
                           // Load-store
                           Constraint_WBSUPPRESS,
                           Constraint_FAULT,
                           Constraint_LIMITED_ATOMICITY, // Accesses are not
                                                         // single-copy atomic
                                                         // above the byte level
                           Constraint_NVNV1_00,
                           Constraint_NVNV1_01,
                           Constraint_NVNV1_11,
                           Constraint_EL1TIMESTAMP,      // Constrain to Virtual Timestamp
                           Constraint_EL2TIMESTAMP,      // Constrain to Virtual Timestamp
                           Constraint_OSH,               // Constrain to Outer Shareable
                           Constraint_ISH,               // Constrain to Inner Shareable
                           Constraint_NSH,               // Constrain to Nonshareable

                           Constraint_NC,                // Constrain to Noncacheable
                           Constraint_WT,                // Constrain to Writethrough
                           Constraint_WB,                // Constrain to Writeback

                           // IPA too large
                           Constraint_FORCE, Constraint_FORCENOSLCHECK,
                           // An unallocated System register value maps onto an allocated value
                           Constraint_MAPTOALLOCATED,
                           // PMSCR_PCT reserved values select Virtual timestamp
                           Constraint_PMSCR_PCT_VIRT
};
```

Library pseudocode for shared/functions/unpredictable/Unpredictable

```
// Unpredictable
// =============
// List of Constrained Unpredictable situations.

enumeration Unpredictable {
                           // VMSR on MVFR
                           Unpredictable_VMSR,
                           // Writeback/transfer register overlap (load)
                           Unpredictable_WBOVERLAPLD,
                           // Writeback/transfer register overlap (store)
                           Unpredictable_WBOVERLAPST,
                           // Load Pair transfer register overlap
                           Unpredictable_LDPOVERLAP,
                           // Store-exclusive base/status register overlap
                           Unpredictable_BASEOVERLAP,
                           // Store-exclusive data/status register overlap
                           Unpredictable_DATAOVERLAP,
                           // Load-store alignment checks
                           Unpredictable_DEVPAGE2,
                           // Instruction fetch from Device memory
                           Unpredictable_INSTRDEVICE,
                           // Reserved CPACR value
                           Unpredictable_RESCPACR,
                           // Reserved MAIR value
                           Unpredictable_RESMAIR,
                           // Effect of SCTLR_ELx.C on Tagged attribute
                           Unpredictable_S1CTAGGED,
                           // Reserved Stage 2 MemAttr value
                           Unpredictable_S2RESMEMATTR,
                           // Reserved TEX:C:B value
                           Unpredictable_RESTEXCB,
                           // Reserved PRRR value
                           Unpredictable_RESPRRR,
                           // Reserved DACR field
                           Unpredictable_RESDACR,
                           // Reserved VTCR.S value
                           Unpredictable_RESVTCRS,
                           // Reserved TCR.TnSZ value
                           Unpredictable_RESTnSZ,
                           // Reserved SCTLR_ELx.TCF value
                           Unpredictable_RESTCF,
                           // Tag stored to Device memory
                           Unpredictable_DEVICETAGSTORE,
                           // Out-of-range TCR.TnSZ value
                           Unpredictable_OORTnSZ,

                           // IPA size exceeds PA size
                           Unpredictable_LARGEIPA,
                           // Syndrome for a known-passing conditional A32 instruction
                           Unpredictable_ESRCONDPASS,
                           // Illegal State exception: zero PSTATE.IT
                           Unpredictable_ILZEROIT,
                           // Illegal State exception: zero PSTATE.T
                           Unpredictable_ILZEROT,
                           // Debug: prioritization of Vector Catch
                           Unpredictable_BPVECTORCATCHPRI,
                           // Debug Vector Catch: match on 2nd halfword
                           Unpredictable_VCMATCHHALF,
                           // Debug Vector Catch: match on Data Abort
                           // or Prefetch abort
                           Unpredictable_VCMATCHDAPA,
                           // Debug watchpoints: nonzero MASK and non-ones BAS
                           Unpredictable_WPMASKANDBAS,
                           // Debug watchpoints: non-contiguous BAS
                           Unpredictable_WPBASCONTIGUOUS,
                           // Debug watchpoints: reserved MASK
                           Unpredictable_RESWPMASK,
                           // Debug watchpoints: nonzero MASKed bits of address
                           Unpredictable_WPMASKEDBITS,
                           // Debug breakpoints and watchpoints: reserved control bits
```

```
                    Unpredictable_RESBPWPCTRL,
                    // Debug breakpoints: not implemented
                    Unpredictable_BPNOTIMPL,
                    // Debug breakpoints: reserved type
                    Unpredictable_RESBPTYPE,
                    // Debug breakpoints and watchpoints: reserved MDSELR_EL1.BANK
                    Unpredictable_RESMDSELR,
                    // Debug breakpoints: not-context-aware breakpoint
                    Unpredictable_BPNOTCTXCMP,
                    // Debug breakpoints: match on 2nd halfword of instruction
                    Unpredictable_BPMATCHHALF,
                    // Debug breakpoints: mismatch on 2nd halfword of instruction
                    Unpredictable_BPMISMATCHHALF,
                    // Debug breakpoints: a breakpoint is linked to that is not
                    // programmed with linking enabled
                    Unpredictable_BPLINKINGDISABLED,
                    // Debug breakpoints: reserved MASK
                    Unpredictable_RESBPMASK,
                    // Debug breakpoints: MASK is set for a Context matching
                    // breakpoint or when DBGBCR_EL1[n].BAS != '1111'
                    Unpredictable_BPMASK,
                    // Debug breakpoints: non-zero MASKed bits of address
                    Unpredictable_BPMASKEDBITS,
                    // Debug breakpoints: A linked breakpoint is
                    // linked to an address matching breakpoint
                    Unpredictable_BPLINKEDADDRMATCH,
                    // Debug: restart to a misaligned AArch32 PC value
                    Unpredictable_RESTARTALIGNPC,
                    // Debug: restart to a not-zero-extended AArch32 PC value
                    Unpredictable_RESTARTZEROUPPERPC,
                    // Zero top 32 bits of X registers in AArch32 state
                    Unpredictable_ZEROUPPER,
                    // Zero top 32 bits of PC on illegal return to
                    // AArch32 state
                    Unpredictable_ERETZEROUPPERPC,
                    // Force address to be aligned when interworking
                    // branch to A32 state
                    Unpredictable_A32FORCEALIGNPC,
                    // SMC disabled
                    Unpredictable_SMD,
                    // FF speculation
                    Unpredictable_NONFAULT,
                    // Zero top bits of Z registers in EL change
                    Unpredictable_SVEZEROUPPER,
                    // Load mem data in NF loads
                    Unpredictable_SVELDNFDATA,
                    // Write zeros in NF loads
                    Unpredictable_SVELDNFZERO,
                    // SP alignment fault when predicate is all zero
                    Unpredictable_CHECKSPNONEACTIVE,
                    // Zero top bits of ZA registers in EL change
                    Unpredictable_SMEZEROUPPER,
                    // HCR_EL2.<NV,NV1> == '01'
                    Unpredictable_NVNV1,
                    // Reserved shareability encoding
                    Unpredictable_Shareability,
                    // Access Flag Update by HW
                    Unpredictable_AFUPDATE,
                    // Dirty Bit State Update by HW
                    Unpredictable_DBUPDATE,
                    // Consider SCTLR[].IESB in Debug state
                    Unpredictable_IESBinDebug,
                    // Bad settings for PMSFCR_EL1/PMSEVFR_EL1/PMSLATFR_EL1
                    Unpredictable_BADPMSFCR,
                    // Zero saved BType value in SPSR_ELx/DPSR_EL0
                    Unpredictable_ZEROBTYPE,
                    // Timestamp constrained to virtual or physical
                    Unpredictable_EL2TIMESTAMP,
                    Unpredictable_EL1TIMESTAMP,
                     // Reserved MDCR_EL3.<NSTBE,NSTB> or MDCR_EL3.<NSPBE,NSPB> value
```

```
                                Unpredictable_RESERVEDNSxB,
                               // WFET or WFIT instruction in Debug state
                               Unpredictable_WFxTDEBUG,
                               // Address does not support LS64 instructions
                               Unpredictable_LS64UNSUPPORTED,
                               // Misaligned exclusives, atomics, acquire/release
                               // to region that is not Normal Cacheable WB
                               Unpredictable_MISALIGNEDATOMIC,
                               // Clearing DCC/ITR sticky flags when instruction is in flight
                               Unpredictable_CLEARERRITEZERO,
                               // ALUEXCEPTIONRETURN when in user/system mode in
                               // A32 instructions
                               Unpredictable_ALUEXCEPTIONRETURN,
                               // Trap to register in debug state are ignored
                               Unpredictable_IGNORETRAPINDEBUG,
                               // Compare DBGBVR.RESS for BP/WP
                               Unpredictable_DBGxVR_RESS,
                               // Inaccessible event counter
                               Unpredictable_PMUEVENTCOUNTER,
                               // Reserved PMSCR.PCT behavior
                               Unpredictable_PMSCR_PCT,
                               // MDCR_EL2.HPMN or HDCR.HPMN is larger than PMCR.N or
                               // FEAT_HPMN0 is not implemented and HPMN is 0.
                               Unpredictable_CounterReservedForEL2,
                               // Generate BRB_FILTRATE event on BRB injection
                               Unpredictable_BRBFILTRATE,
                               // Operands for CPY*/SET* instructions overlap or
                               // use 0b11111 as a register specifier
                               Unpredictable_MOPSOVERLAP31,
                               // Store-only Tag checking on a failed Atomic Compare and Swap
                               Unpredictable_STOREONLYTAGCHECKEDCAS,
                               // Reserved MDCR_EL3.ETBAD value
                               Unpredictable_RES_ETBAD,
                               // Reserved PMEVTYPER<n>_EL0.TC value
                               Unpredictable_RESTC
};
```

Library pseudocode for shared/functions/vector/AdvSIMDExpandImm

```
// AdvSIMDExpandImm()
// ==================

bits(64) AdvSIMDExpandImm(bit op, bits(4) cmode, bits(8) imm8)
    bits(64) imm64;
    case cmode<3:1> of
        when '000'
            imm64 = Replicate(Zeros(24):imm8, 2);
        when '001'
            imm64 = Replicate(Zeros(16):imm8:Zeros(8), 2);
        when '010'
            imm64 = Replicate(Zeros(8):imm8:Zeros(16), 2);
        when '011'
            imm64 = Replicate(imm8:Zeros(24), 2);
        when '100'
            imm64 = Replicate(Zeros(8):imm8, 4);
        when '101'
            imm64 = Replicate(imm8:Zeros(8), 4);
        when '110'
            if cmode<0> == '0' then
                imm64 = Replicate(Zeros(16):imm8:Ones(8), 2);
            else
                imm64 = Replicate(Zeros(8):imm8:Ones(16), 2);
        when '111'
            if cmode<0> == '0' && op == '0' then
                imm64 = Replicate(imm8, 8);
            if cmode<0> == '0' && op == '1' then
                imm8a = Replicate(imm8<7>, 8); imm8b = Replicate(imm8<6>, 8);
                imm8c = Replicate(imm8<5>, 8); imm8d = Replicate(imm8<4>, 8);
                imm8e = Replicate(imm8<3>, 8); imm8f = Replicate(imm8<2>, 8);
                imm8g = Replicate(imm8<1>, 8); imm8h = Replicate(imm8<0>, 8);
                imm64 = imm8a:imm8b:imm8c:imm8d:imm8e:imm8f:imm8g:imm8h;
            if cmode<0> == '1' && op == '0' then
                imm32 = imm8<7>:NOT(imm8<6>):Replicate(imm8<6>,5):imm8<5:0>:Zeros(19);
                imm64 = Replicate(imm32, 2);
            if cmode<0> == '1' && op == '1' then
                if UsingAArch32() then ReservedEncoding();
                imm64 = imm8<7>:NOT(imm8<6>):Replicate(imm8<6>,8):imm8<5:0>:Zeros(48);

    return imm64;
```

Library pseudocode for shared/functions/vector/MatMulAdd

```
// MatMulAdd()
// ===========
//
// Signed or unsigned 8-bit integer matrix multiply and add to 32-bit integer matrix
// result[2, 2] = addend[2, 2] + (op1[2, 8] * op2[8, 2])

bits(N) MatMulAdd(bits(N) addend, bits(N) op1, bits(N) op2, boolean op1_unsigned,
                  boolean op2_unsigned)
    assert N == 128;

    bits(N)  result;
    bits(32) sum;
    integer  prod;

    for i = 0 to 1
        for j = 0 to 1
            sum = Elem[addend, 2*i + j, 32];
            for k = 0 to 7
                prod = (Int(Elem[op1, 8*i + k, 8], op1_unsigned) *
                        Int(Elem[op2, 8*j + k, 8], op2_unsigned));
                sum  = sum + prod;
            Elem[result, 2*i + j, 32] = sum;

    return result;
```

Library pseudocode for shared/functions/vector/PolynomialMult

```
// PolynomialMult()
// ================

bits(M+N) PolynomialMult(bits(M) op1, bits(N) op2)
    result = Zeros(M+N);
    extended_op2 = ZeroExtend(op2, M+N);
    for i=0 to M-1
        if op1<i> == '1' then
            result = result EOR LSL(extended_op2, i);
    return result;
```

Library pseudocode for shared/functions/vector/SatQ

```
// SatQ()
// ======

(bits(N), boolean) SatQ(integer i, integer N, boolean unsigned)
    (result, sat) = if unsigned then UnsignedSatQ(i, N) else SignedSatQ(i, N);
    return (result, sat);
```

Library pseudocode for shared/functions/vector/SignedSatQ

```
// SignedSatQ()
// ============

(bits(N), boolean) SignedSatQ(integer i, integer N)
    integer result;
    boolean saturated;
    if i > 2^(N-1) - 1 then
        result = 2^(N-1) - 1;  saturated = TRUE;
    elsif i < -(2^(N-1)) then
        result = -(2^(N-1));  saturated = TRUE;
    else
        result = i;  saturated = FALSE;
    return (result<N-1:0>, saturated);
```

Library pseudocode for shared/functions/vector/UnsignedRSqrtEstimate

```
// UnsignedRSqrtEstimate()
// =======================

bits(N) UnsignedRSqrtEstimate(bits(N) operand)
    assert N == 32;
    bits(N) result;
    if operand<N-1:N-2> == '00' then  // Operands <= 0x3FFFFFFF produce 0xFFFFFFFF
        result = Ones(N);
    else
        // input is in the range 0x40000000 .. 0xffffffff representing [0.25 .. 1.0)
        // estimate is in the range 256 .. 511 representing [1.0 .. 2.0)
        increasedprecision = FALSE;
        estimate = RecipSqrtEstimate(UInt(operand<31:23>), increasedprecision);
        // result is in the range 0x80000000 .. 0xff800000 representing [1.0 .. 2.0)
        result = estimate<8:0> : Zeros(N-9);

    return result;
```

Library pseudocode for shared/functions/vector/UnsignedRecipEstimate

```
// UnsignedRecipEstimate()
// =======================

bits(N) UnsignedRecipEstimate(bits(N) operand)
    assert N == 32;
    bits(N) result;
    if operand<N-1> == '0' then  // Operands <= 0x7FFFFFFF produce 0xFFFFFFFF
        result = Ones(N);
    else
        // input is in the range 0x80000000 .. 0xffffffff representing [0.5 .. 1.0)

        // estimate is in the range 256 to 511 representing [1.0 .. 2.0)
        increasedprecision = FALSE;
        estimate = RecipEstimate(UInt(operand<31:23>), increasedprecision);

        // result is in the range 0x80000000 .. 0xff800000 representing [1.0 .. 2.0)
        result = estimate<8:0> : Zeros(N-9);

    return result;
```

Library pseudocode for shared/functions/vector/UnsignedSatQ

```
// UnsignedSatQ()
// ==============

(bits(N), boolean) UnsignedSatQ(integer i, integer N)
    integer result;
    boolean saturated;
    if i > 2^N - 1 then
        result = 2^N - 1;  saturated = TRUE;
    elsif i < 0 then
        result = 0;  saturated = TRUE;
    else
        result = i;  saturated = FALSE;
    return (result<N-1:0>, saturated);
```

Library pseudocode for shared/trace/Common/DebugMemWrite

```
// DebugMemWrite()
// ===============
// Write data to memory one byte at a time. Starting at the passed virtual address.
// Used by SPE.

(PhysMemRetStatus, AddressDescriptor) DebugMemWrite(bits(64) vaddress, AccessDescriptor accdesc,
                                                    boolean aligned, bits(8) data)

    PhysMemRetStatus memstatus = PhysMemRetStatus UNKNOWN;

    // Translate virtual address
    AddressDescriptor addrdesc;
    integer size = 1;
    addrdesc = AArch64.TranslateAddress(vaddress, accdesc, aligned, size);

    if IsFault(addrdesc) then
        return (memstatus, addrdesc);

    memstatus = PhysMemWrite(addrdesc, 1, accdesc, data);

    return (memstatus, addrdesc);
```

Library pseudocode for shared/trace/Common/DebugWriteExternalAbort

```
// DebugWriteExternalAbort()
// =========================
// Populate the syndrome register for an External abort caused by a call of DebugMemWrite().

DebugWriteExternalAbort(PhysMemRetStatus memstatus, AddressDescriptor addrdesc,
                        bits(64) start_vaddr)

    boolean iswrite = TRUE;

    boolean handle_as_SError = FALSE;
    boolean async_external_abort = FALSE;
    bits(64) syndrome;
    case addrdesc.fault.access.acctype of
        when AccessType_SPE
            handle_as_SError = boolean IMPLEMENTATION_DEFINED "SPE SyncExternal as SError";
            async_external_abort = boolean IMPLEMENTATION_DEFINED "SPE async External abort";
            syndrome = PMBSR_EL1<63:0>;
        otherwise
            Unreachable();

    boolean ttw_abort;
    ttw_abort = addrdesc.fault.statuscode IN {Fault_SyncExternalOnWalk,
                                              Fault_SyncParityOnWalk};
    Fault statuscode = if ttw_abort then addrdesc.fault.statuscode else memstatus.statuscode;
    bit extflag = if ttw_abort then addrdesc.fault.extflag else memstatus.extflag;
    if (statuscode IN {Fault_AsyncExternal, Fault_AsyncParity} || handle_as_SError) then
        // ASYNC Fault -> SError or SYNC Fault handled as SError
        FaultRecord fault = NoFault();
        boolean parity = statuscode IN {Fault_SyncParity, Fault_AsyncParity,
                                        Fault_SyncParityOnWalk};
        fault.statuscode = if parity then Fault_AsyncParity else Fault_AsyncExternal;
        if HaveRASExt() then
            fault.merrorstate = memstatus.merrorstate;
        fault.extflag = extflag;
        fault.access.acctype = addrdesc.fault.access.acctype;
        PendSErrorInterrupt(fault);
    else
        // SYNC Fault, not handled by SError
        // Generate Buffer Management Event
        // EA bit
        syndrome<18> = '1';

        // DL bit for SPE
        if addrdesc.fault.access.acctype == AccessType_SPE && (async_external_abort ||
            (start_vaddr != addrdesc.vaddress)) then
            syndrome<19> = '1';

        // Do not change following values if previous Buffer Management Event
        // has not been handled.
        // S bit
        if IsZero(syndrome<17>) then
            syndrome<17> = '1';

            // EC bits
            bits(6) ec;
            if (HaveRME() && addrdesc.fault.gpcf.gpf != GPCF_None &&
                addrdesc.fault.gpcf.gpf != GPCF_Fail) then
                ec = '011110';
            else
                ec = if addrdesc.fault.secondstage then '100101' else '100100';
            syndrome<31:26> = ec;

            // MSS bits
            if async_external_abort then
                syndrome<15:0> = Zeros(10) : '010001';
            else
                syndrome<15:0> = Zeros(10) : EncodeLDFSC(statuscode, addrdesc.fault.level);

        case addrdesc.fault.access.acctype of
            when AccessType_SPE
```

```
            PMBSR_EL1<63:0> = syndrome;
        otherwise
            Unreachable();
```

Library pseudocode for shared/trace/Common/DebugWriteFault

```
// DebugWriteFault()
// =================
// Populate the syndrome register for a Translation fault caused by a call of DebugMemWrite().

DebugWriteFault(bits(64) vaddress, FaultRecord fault)
    bits(64) syndrome;
    case fault.access.acctype of
        when AccessType_SPE
            syndrome = PMBSR_EL1<63:0>;
        otherwise
            Unreachable();

    // MSS
    syndrome<15:0> = Zeros(10) : EncodeLDFSC(fault.statuscode, fault.level);

    // MSS2
    syndrome<55:32> = Zeros(24);

    // EC bits
    bits(6) ec;
    if HaveRME() && fault.gpcf.gpf != GPCF_None && fault.gpcf.gpf != GPCF_Fail then
        ec = '011110';
    else
        ec = if fault.secondstage then '100101' else '100100';
    syndrome<31:26> = ec;

    // S bit
    syndrome<17> = '1';

    if fault.statuscode == Fault_Permission then
        // assuredonly bit
        syndrome<39> = if fault.assuredonly then '1' else '0';
        // overlay bit
        syndrome<38> = if fault.overlay then '1' else '0';
        // dirtybit
        syndrome<37> = if fault.dirtybit then '1' else '0';

    case fault.access.acctype of
        when AccessType_SPE
            PMBSR_EL1<63:0> = syndrome;
        otherwise
            Unreachable();

    // Buffer Write Pointer already points to the address that generated the fault.
    // Writing to memory never started so no data loss. DL is unchanged.

    return;
```

Library pseudocode for shared/trace/Common/GetTimestamp

```
// GetTimestamp()
// ==============
// Returns the Timestamp depending on the type

bits(64) GetTimestamp(TimeStamp timeStampType)
    case timeStampType of
        when TimeStamp_Physical
            return PhysicalCountInt();
        when TimeStamp_Virtual
            return PhysicalCountInt() - CNTVOFF_EL2;
        when TimeStamp_OffsetPhysical
            bits(64) physoff = if PhysicalOffsetIsValid() then CNTPOFF_EL2 else Zeros(64);
            return PhysicalCountInt() - physoff;
        when TimeStamp_None
            return Zeros(64);
        when TimeStamp_CoreSight
            return bits(64) IMPLEMENTATION_DEFINED "CoreSight timestamp";
        otherwise
            Unreachable();
```

Library pseudocode for shared/trace/Common/PhysicalOffsetIsValid

```
// PhysicalOffsetIsValid()
// =======================
// Returns whether the Physical offset for the timestamp is valid

boolean PhysicalOffsetIsValid()
    if !HaveAArch64() then
        return FALSE;
    elsif !HaveEL(EL2) || !HaveECVExt() then
        return FALSE;
    elsif HaveEL(EL3) && SCR_EL3.NS == '1' && EffectiveSCR_EL3_RW() == '0' then
        return FALSE;
    elsif HaveEL(EL3) && SCR_EL3.ECVEn == '0' then
        return FALSE;
    elsif CNTHCTL_EL2.ECV == '0' then
        return FALSE;
    else
        return TRUE;
```

Library pseudocode for shared/trace/TraceBranch/BranchNotTaken

```
// BranchNotTaken()
// ================
// Called when a branch is not taken.

BranchNotTaken(BranchType branchtype, boolean branch_conditional)
    boolean branchtaken = FALSE;
    if HaveStatisticalProfiling() then
        SPEBranch(bits(64) UNKNOWN, branchtype, branch_conditional, branchtaken);
    return;
```

Library pseudocode for shared/trace/TraceBuffer/AllowExternalTraceBufferAccess

```
// AllowExternalTraceBufferAccess()
// ================================
// Returns TRUE if an external debug interface access to the Trace Buffer
// registers is allowed, FALSE otherwise.
// The access may also be subject to OS Lock, power-down, etc.

boolean AllowExternalTraceBufferAccess()
    return AllowExternalTraceBufferAccess(AccessState());

// AllowExternalTraceBufferAccess()
// ================================
// Returns TRUE if an external debug interface access to the Trace Buffer
// registers is allowed for the given Security state, FALSE otherwise.
// The access may also be subject to OS Lock, power-down, etc.

boolean AllowExternalTraceBufferAccess(SecurityState access_state)
    assert IsFeatureImplemented(FEAT_TRBE_EXT);
    assert IsFeatureImplemented(FEAT_Debugv8p4);  // Required when Trace Buffer implemented

    bits(2) etbad = if HaveEL(EL3) then MDCR_EL3.ETBAD else '11';

    // Check for reserved values
    if !IsFeatureImplemented(FEAT_RME) && etbad IN {'01','10'} then
        Constraint c;
        (c, etbad) = ConstrainUnpredictableBits(Unpredictable_RES_ETBAD, 2);
        assert c IN {Constraint_DISABLED, Constraint_UNKNOWN};
        if c == Constraint_DISABLED then return FALSE;
        // Otherwise the value returned by ConstrainUnpredictableBits must be a
        // not-reserved value

    case etbad of
        when '00'
            SecurityState ss = (if IsFeatureImplemented(FEAT_RME) then SS_Root else SS_Secure);
            return access_state == ss;
        when '01'
            return access_state IN {SS_Root, SS_Realm};
        when '10'
            return access_state IN {SS_Root, SS_Secure};
        when '11'
            return TRUE;
```

Library pseudocode for shared/trace/TraceBuffer/TraceBufferEnabled

```
// TraceBufferEnabled()
// ====================

boolean TraceBufferEnabled()
    if !HaveTraceBufferExtension() || TRBLIMITR_EL1.E == '0' then
        return FALSE;
    if !SelfHostedTraceEnabled() then
        return FALSE;
    (-, el) = TraceBufferOwner();
    return !ELUsingAArch32(el);
```

Library pseudocode for shared/trace/TraceBuffer/TraceBufferOwner

```
// TraceBufferOwner()
// ==================
// Return the owning Security state and Exception level. Must only be called
// when SelfHostedTraceEnabled() is TRUE.

(SecurityState, bits(2)) TraceBufferOwner()
    assert HaveTraceBufferExtension() && SelfHostedTraceEnabled();

    SecurityState owning_ss;
    if HaveEL(EL3) then
        bits(3) state_bits;
        if HaveRME() then
            state_bits = MDCR_EL3.<NSTBE,NSTB>;
            if (state_bits IN {'10x'} ||
                (!HaveSecureEL2Ext() && state_bits IN {'00x'})) then
                // Reserved value
                (-, state_bits) = ConstrainUnpredictableBits(Unpredictable_RESERVEDNSxB, 3);
        else
            state_bits = '0' : MDCR_EL3.NSTB;

        case state_bits of
            when '00x' owning_ss = SS_Secure;
            when '01x' owning_ss = SS_NonSecure;
            when '11x' owning_ss = SS_Realm;
    else
        owning_ss = if SecureOnlyImplementation() then SS_Secure else SS_NonSecure;
    bits(2) owning_el;
    if HaveEL(EL2) && (owning_ss != SS_Secure || IsSecureEL2Enabled()) then
        owning_el = if MDCR_EL2.E2TB == '00' then EL2 else EL1;
    else
        owning_el = EL1;
    return (owning_ss, owning_el);
```

Library pseudocode for shared/trace/TraceBuffer/TraceBufferRunning

```
// TraceBufferRunning()
// ====================

boolean TraceBufferRunning()
    return TraceBufferEnabled() && TRBSR_EL1.S == '0';
```

Library pseudocode for shared/trace/TraceInstrumentationAllowed/TraceInstrumentationAllowed

```
// TraceInstrumentationAllowed()
// =============================
// Returns TRUE if Instrumentation Trace is allowed
// in the given Exception level and Security state.

boolean TraceInstrumentationAllowed(SecurityState ss, bits(2) el)
    if !IsFeatureImplemented(FEAT_ITE) then return FALSE;
    if ELUsingAArch32(el) then return FALSE;

    if TraceAllowed(el) then
        bit ite_bit;
        case el of
            when EL3 ite_bit = '0';
            when EL2 ite_bit = TRCITECR_EL2.E2E;
            when EL1 ite_bit = TRCITECR_EL1.E1E;
            when EL0
                if EffectiveTGE() == '1' then
                    ite_bit = TRCITECR_EL2.E0HE;
                else
                    ite_bit = TRCITECR_EL1.E0E;

        if SelfHostedTraceEnabled() then
            return ite_bit == '1';
        else
            bit el_bit;
            bit ss_bit;
            case el of
                when EL0 el_bit = TRCITEEDCR.E0;
                when EL1 el_bit = TRCITEEDCR.E1;
                when EL2 el_bit = TRCITEEDCR.E2;
                when EL3 el_bit = TRCITEEDCR.E3;
            case ss of
                when SS_Realm     ss_bit = TRCITEEDCR.RL;
                when SS_Secure    ss_bit = TRCITEEDCR.S;
                when SS_NonSecure ss_bit = TRCITEEDCR.NS;
                otherwise ss_bit = '1';

            boolean ed_allowed = ss_bit == '1' && el_bit == '1';

            if TRCCONFIGR.ITO == '1' then
                return ed_allowed;
            else
                return ed_allowed && ite_bit == '1';
    else
        return FALSE;
```

Library pseudocode for shared/trace/selfhosted/EffectiveE0HTRE

```
// EffectiveE0HTRE()
// =================
// Returns effective E0HTRE value

bit EffectiveE0HTRE()
    return if ELUsingAArch32(EL2) then HTRFCR.E0HTRE else TRFCR_EL2.E0HTRE;
```

Library pseudocode for shared/trace/selfhosted/EffectiveE0TRE

```
// EffectiveE0TRE()
// ================
// Returns effective E0TRE value

bit EffectiveE0TRE()
    return if ELUsingAArch32(EL1) then TRFCR.E0TRE else TRFCR_EL1.E0TRE;
```

Library pseudocode for shared/trace/selfhosted/EffectiveE1TRE

```
// EffectiveE1TRE()
// ================
// Returns effective E1TRE value

bit EffectiveE1TRE()
    return if UsingAArch32() then TRFCR.E1TRE else TRFCR_EL1.E1TRE;
```

Library pseudocode for shared/trace/selfhosted/EffectiveE2TRE

```
// EffectiveE2TRE()
// ================
// Returns effective E2TRE value

bit EffectiveE2TRE()
    return if UsingAArch32() then HTRFCR.E2TRE else TRFCR_EL2.E2TRE;
```

Library pseudocode for shared/trace/selfhosted/SelfHostedTraceEnabled

```
// SelfHostedTraceEnabled()
// ========================
// Returns TRUE if Self-hosted Trace is enabled.

boolean SelfHostedTraceEnabled()
    bit secure_trace_enable = '0';
    if !(HaveTraceExt() && HaveSelfHostedTrace()) then return FALSE;
    if EDSCR.TFO == '0' then return TRUE;
    if HaveRME() then
        secure_trace_enable = if HaveSecureEL2Ext() then MDCR_EL3.STE else '0';
        return ((secure_trace_enable == '1' && !ExternalSecureNoninvasiveDebugEnabled()) ||
                (MDCR_EL3.RLTE == '1' && !ExternalRealmNoninvasiveDebugEnabled()));
    if HaveEL(EL3) then
        secure_trace_enable = if ELUsingAArch32(EL3) then SDCR.STE else MDCR_EL3.STE;
    else
        secure_trace_enable = if SecureOnlyImplementation() then '1' else '0';

    if secure_trace_enable == '1' && !ExternalSecureNoninvasiveDebugEnabled() then
        return TRUE;

    return FALSE;
```

Library pseudocode for shared/trace/selfhosted/TraceAllowed

```
// TraceAllowed()
// ==============
// Returns TRUE if Self-hosted Trace is allowed in the given Exception level.

boolean TraceAllowed(bits(2) el)
    if !HaveTraceExt() then return FALSE;
    if SelfHostedTraceEnabled() then
        boolean trace_allowed;
        ss = SecurityStateAtEL(el);
        // Detect scenarios where tracing in this Security state is never allowed.
        case ss of
            when SS_NonSecure
                trace_allowed = TRUE;
            when SS_Secure
                bit trace_bit;
                if HaveEL(EL3) then
                    trace_bit = if ELUsingAArch32(EL3) then SDCR.STE else MDCR_EL3.STE;
                else
                    trace_bit = '1';
                trace_allowed = trace_bit == '1';
            when SS_Realm
                trace_allowed = MDCR_EL3.RLTE == '1';
            when SS_Root
                trace_allowed = FALSE;

        // Tracing is prohibited if the trace buffer owning security state is not the
        // current Security state or the owning Exception level is a lower Exception level.
        if HaveTraceBufferExtension() && TraceBufferEnabled() then
            (owning_ss, owning_el) = TraceBufferOwner();
            if (ss != owning_ss || UInt(owning_el) < UInt(el) ||
                (EffectiveTGE() == '1' && owning_el == EL1)) then
                trace_allowed = FALSE;

        bit TRE_bit;
        case el of
            when EL3  TRE_bit = if !HaveAArch64() then TRFCR.E1TRE else '0';
            when EL2  TRE_bit = EffectiveE2TRE();
            when EL1  TRE_bit = EffectiveE1TRE();
            when EL0
                if EffectiveTGE() == '1' then
                    TRE_bit = EffectiveE0HTRE();
                else
                    TRE_bit = EffectiveE0TRE();

        return trace_allowed && TRE_bit == '1';
    else
        return ExternalNoninvasiveDebugAllowed(el);
```

Library pseudocode for shared/trace/selfhosted/TraceContextIDR2

```
// TraceContextIDR2()
// ==================

boolean TraceContextIDR2()
    if !TraceAllowed(PSTATE.EL)|| !HaveEL(EL2) then return FALSE;
    return (!SelfHostedTraceEnabled() || TRFCR_EL2.CX == '1');
```

Library pseudocode for shared/trace/selfhosted/TraceSynchronizationBarrier

```
// TraceSynchronizationBarrier()
// =============================
// Memory barrier instruction that preserves the relative order of memory accesses to System
// registers due to trace operations and other memory accesses to the same registers

TraceSynchronizationBarrier();
```

Library pseudocode for shared/trace/selfhosted/TraceTimeStamp

```
// TraceTimeStamp()
// ================

TimeStamp TraceTimeStamp()
    if SelfHostedTraceEnabled() then
        if HaveEL(EL2) then
            TS_el2 = TRFCR_EL2.TS;
            if !HaveECVExt() && TS_el2 == '10' then
                // Reserved value
                (-, TS_el2) = ConstrainUnpredictableBits(Unpredictable_EL2TIMESTAMP, 2);

            case TS_el2 of
                when '00'
                    // Falls out to check TRFCR_EL1.TS
                when '01'
                    return TimeStamp_Virtual;
                when '10'
                    assert HaveECVExt();  // Otherwise ConstrainUnpredictableBits removes this case
                    return TimeStamp_OffsetPhysical;
                when '11'
                    return TimeStamp_Physical;

        TS_el1 = TRFCR_EL1.TS;
        if TS_el1 == '00' || (!HaveECVExt() && TS_el1 == '10') then
            // Reserved value
            (-, TS_el1) = ConstrainUnpredictableBits(Unpredictable_EL1TIMESTAMP, 2);

        case TS_el1 of
            when '01'
                return TimeStamp_Virtual;
            when '10'
                assert HaveECVExt();
                return TimeStamp_OffsetPhysical;
            when '11'
                return TimeStamp_Physical;
            otherwise
                Unreachable();         // ConstrainUnpredictableBits removes this case
    else
        return TimeStamp_CoreSight;
```

Library pseudocode for shared/trace/system/IsTraceCorePowered

```
// IsTraceCorePowered()
// ====================
// Returns TRUE if the Trace Core Power Domain is powered up

boolean IsTraceCorePowered();
```

Library pseudocode for shared/translation/at

```
enumeration TranslationStage {
    TranslationStage_1,
    TranslationStage_12
};

enumeration ATAccess {
    ATAccess_Read,
    ATAccess_Write,
    ATAccess_ReadPAN,
    ATAccess_WritePAN
};
```

Library pseudocode for shared/translation/at/EncodePARAttrs

```
// EncodePARAttrs()
// ================
// Convert orthogonal attributes and hints to 64-bit PAR ATTR field.

bits(8) EncodePARAttrs(MemoryAttributes memattrs)
    bits(8) result;

    if HaveMTEExt() && memattrs.tags == MemTag_AllocationTagged then
        if HaveMTEPermExt() && memattrs.notagaccess then
            result<7:0> = '11100000';
        else
            result<7:0> = '11110000';
        return result;

    if memattrs.memtype == MemType_Device then
        result<7:4> = '0000';
        case memattrs.device of
            when DeviceType_nGnRnE result<3:0> = '0000';
            when DeviceType_nGnRE  result<3:0> = '0100';
            when DeviceType_nGRE   result<3:0> = '1000';
            when DeviceType_GRE    result<3:0> = '1100';
            otherwise              Unreachable();
        result<0> = NOT memattrs.xs;
    else
        if memattrs.xs == '0' then
            if (memattrs.outer.attrs == MemAttr_WT && memattrs.inner.attrs == MemAttr_WT &&
                    !memattrs.outer.transient && memattrs.outer.hints == MemHint_RA) then
                return '10100000';
            elsif memattrs.outer.attrs == MemAttr_NC && memattrs.inner.attrs == MemAttr_NC then
                return '01000000';

        if memattrs.outer.attrs == MemAttr_WT then
            result<7:6> = if memattrs.outer.transient then '00' else '10';
            result<5:4> = memattrs.outer.hints;
        elsif memattrs.outer.attrs == MemAttr_WB then
            result<7:6> = if memattrs.outer.transient then '01' else '11';
            result<5:4> = memattrs.outer.hints;
        else // MemAttr_NC
            result<7:4> = '0100';

        if memattrs.inner.attrs == MemAttr_WT then
            result<3:2> = if memattrs.inner.transient then '00' else '10';
            result<1:0> = memattrs.inner.hints;
        elsif memattrs.inner.attrs == MemAttr_WB then
            result<3:2> = if memattrs.inner.transient then '01' else '11';
            result<1:0> = memattrs.inner.hints;
        else // MemAttr_NC
            result<3:0> = '0100';

    return result;
```

Library pseudocode for shared/translation/at/PAREncodeShareability

```
// PAREncodeShareability()
// =======================
// Derive 64-bit PAR SH field.

bits(2) PAREncodeShareability(MemoryAttributes memattrs)
    if (memattrs.memtype == MemType_Device ||
            (memattrs.inner.attrs == MemAttr_NC &&
             memattrs.outer.attrs == MemAttr_NC)) then
        // Force Outer-Shareable on Device and Normal Non-Cacheable memory
        return '10';

    case memattrs.shareability of
        when Shareability_NSH return '00';
        when Shareability_ISH return '11';
        when Shareability_OSH return '10';
```

Library pseudocode for shared/translation/at/ReportedPARAttrs

```
// ReportedPARAttrs()
// ==================
// The value returned in this field can be the resulting attribute, as determined by any permitted
// implementation choices and any applicable configuration bits, instead of the value that appears
// in the translation table descriptor.

bits(8) ReportedPARAttrs(bits(8) parattrs);
```

Library pseudocode for shared/translation/at/ReportedPARShareability

```
// ReportedPARShareability()
// =========================
// The value returned in SH field can be the resulting attribute, as determined by any
// permitted implementation choices and any applicable configuration bits, instead of
// the value that appears in the translation table descriptor.

bits(2) ReportedPARShareability(bits(2) sh);
```

Library pseudocode for shared/translation/attrs/DecodeDevice

```
// DecodeDevice()
// ==============
// Decode output Device type

DeviceType DecodeDevice(bits(2) device)
    case device of
        when '00'   return DeviceType_nGnRnE;
        when '01'   return DeviceType_nGnRE;
        when '10'   return DeviceType_nGRE;
        when '11'   return DeviceType_GRE;
```

Library pseudocode for shared/translation/attrs/DecodeLDFAttr

```
// DecodeLDFAttr()
// ===============
// Decode memory attributes using LDF (Long Descriptor Format) mapping

MemAttrHints DecodeLDFAttr(bits(4) attr)
    MemAttrHints ldfattr;

    if    attr IN {'x0xx'} then ldfattr.attrs = MemAttr_WT; // Write-through
    elsif attr == '0100' then ldfattr.attrs = MemAttr_NC;   // Non-cacheable
    elsif attr IN {'x1xx'} then ldfattr.attrs = MemAttr_WB; // Write-back
    else                        Unreachable();

    // Allocation hints are applicable only to cacheable memory.
    if ldfattr.attrs != MemAttr_NC then
        case attr<1:0> of
            when '00' ldfattr.hints = MemHint_No;  // No allocation hints
            when '01' ldfattr.hints = MemHint_WA;  // Write-allocate
            when '10' ldfattr.hints = MemHint_RA;  // Read-allocate
            when '11' ldfattr.hints = MemHint_RWA; // Read/Write allocate

    // The Transient hint applies only to cacheable memory with some allocation hints.
    if ldfattr.attrs != MemAttr_NC && ldfattr.hints != MemHint_No then
        ldfattr.transient = attr<3> == '0';

    return ldfattr;
```

Library pseudocode for shared/translation/attrs/DecodeSDFAttr

```
// DecodeSDFAttr()
// ===============
// Decode memory attributes using SDF (Short Descriptor Format) mapping

MemAttrHints DecodeSDFAttr(bits(2) rgn)
    MemAttrHints sdfattr;

    case rgn of
        when '00'                   // Non-cacheable (no allocate)
            sdfattr.attrs = MemAttr_NC;
        when '01'                   // Write-back, Read and Write allocate
            sdfattr.attrs = MemAttr_WB;
            sdfattr.hints = MemHint_RWA;
        when '10'                   // Write-through, Read allocate
            sdfattr.attrs = MemAttr_WT;
            sdfattr.hints = MemHint_RA;
        when '11'                   // Write-back, Read allocate
            sdfattr.attrs = MemAttr_WB;
            sdfattr.hints = MemHint_RA;

    sdfattr.transient = FALSE;

    return sdfattr;
```

Library pseudocode for shared/translation/attrs/DecodeShareability

```
// DecodeShareability()
// ====================
// Decode shareability of target memory region

Shareability DecodeShareability(bits(2) sh)
    case sh of
        when '10' return Shareability_OSH;
        when '11' return Shareability_ISH;
        when '00' return Shareability_NSH;
        otherwise
            case ConstrainUnpredictable(Unpredictable_Shareability) of
                when Constraint_OSH return Shareability_OSH;
                when Constraint_ISH return Shareability_ISH;
                when Constraint_NSH return Shareability_NSH;
```

Library pseudocode for shared/translation/attrs/EffectiveShareability

```
// EffectiveShareability()
// =======================
// Force Outer Shareability on Device and Normal iNCoNC memory

Shareability EffectiveShareability(MemoryAttributes memattrs)
    if (memattrs.memtype == MemType_Device ||
            (memattrs.inner.attrs == MemAttr_NC &&
             memattrs.outer.attrs == MemAttr_NC)) then
        return Shareability_OSH;
    else
        return memattrs.shareability;
```

Library pseudocode for shared/translation/attrs/NormalNCMemAttr

```
// NormalNCMemAttr()
// =================
// Normal Non-cacheable memory attributes

MemoryAttributes NormalNCMemAttr()
    MemAttrHints non_cacheable;
    non_cacheable.attrs = MemAttr_NC;

    MemoryAttributes nc_memattrs;
    nc_memattrs.memtype      = MemType_Normal;
    nc_memattrs.outer        = non_cacheable;
    nc_memattrs.inner        = non_cacheable;
    nc_memattrs.shareability = Shareability_OSH;
    nc_memattrs.tags         = MemTag_Untagged;
    nc_memattrs.notagaccess  = FALSE;

    return nc_memattrs;
```

Library pseudocode for shared/translation/attrs/S1ConstrainUnpredictableRESMAIR

```
// S1ConstrainUnpredictableRESMAIR()
// =================================
// Determine whether a reserved value occupies MAIR_ELx.AttrN

boolean S1ConstrainUnpredictableRESMAIR(bits(8) attr, boolean s1aarch64)
    case attr of
        when '0000xx01' return !(s1aarch64 && HaveFeatXS());
        when '0000xxxx' return attr<1:0> != '00';
        when '01000000' return !(s1aarch64 && HaveFeatXS());
        when '10100000' return !(s1aarch64 && HaveFeatXS());
        when '11110000' return !(s1aarch64 && HaveMTE2Ext());
        when 'xxxx0000' return TRUE;
        otherwise       return FALSE;
```

Library pseudocode for shared/translation/attrs/S1DecodeMemAttrs

```
// S1DecodeMemAttrs()
// ==================
// Decode MAIR-format memory attributes assigned in stage 1

MemoryAttributes S1DecodeMemAttrs(bits(8) attr_in, bits(2) sh, boolean s1aarch64,
                                  S1TTWParams walkparams)
    bits(8) attr = attr_in;
    if S1ConstrainUnpredictableRESMAIR(attr, s1aarch64) then
        (-, attr) = ConstrainUnpredictableBits(Unpredictable_RESMAIR, 8);

    MemoryAttributes memattrs;
    case attr of
        when '0000xxxx' // Device memory
            memattrs.memtype = MemType_Device;
            memattrs.device  = DecodeDevice(attr<3:2>);
            memattrs.xs      = if s1aarch64 then NOT attr<0> else '1';
        when '01000000'
            assert s1aarch64 && HaveFeatXS();
            memattrs.memtype = MemType_Normal;
            memattrs.outer.attrs = MemAttr_NC;
            memattrs.inner.attrs = MemAttr_NC;
            memattrs.xs          = '0';

        when '10100000'
            assert s1aarch64 && HaveFeatXS();
            memattrs.memtype = MemType_Normal;
            memattrs.outer.attrs     = MemAttr_WT;
            memattrs.outer.hints     = MemHint_RA;
            memattrs.outer.transient = FALSE;
            memattrs.inner.attrs     = MemAttr_WT;
            memattrs.inner.hints     = MemHint_RA;
            memattrs.inner.transient = FALSE;
            memattrs.xs              = '0';
        when '11110000' // Tagged memory
            assert s1aarch64 && HaveMTE2Ext();
            memattrs.memtype = MemType_Normal;
            memattrs.outer.attrs     = MemAttr_WB;
            memattrs.outer.hints     = MemHint_RWA;
            memattrs.outer.transient = FALSE;
            memattrs.inner.attrs     = MemAttr_WB;
            memattrs.inner.hints     = MemHint_RWA;
            memattrs.inner.transient = FALSE;
            memattrs.xs              = '0';
        otherwise
            memattrs.memtype = MemType_Normal;
            memattrs.outer   = DecodeLDFAttr(attr<7:4>);
            memattrs.inner   = DecodeLDFAttr(attr<3:0>);

            if (memattrs.inner.attrs == MemAttr_WB &&
                    memattrs.outer.attrs == MemAttr_WB) then
                memattrs.xs = '0';
            else
                memattrs.xs = '1';

    if s1aarch64 && attr IN {'11110000'} then
        memattrs.tags = MemTag_AllocationTagged;
    elsif s1aarch64 && walkparams.mtx == '1' then
        memattrs.tags = MemTag_CanonicallyTagged;
    else
        memattrs.tags = MemTag_Untagged;

    memattrs.notagaccess = FALSE;

    memattrs.shareability = DecodeShareability(sh);

    return memattrs;
```

Library pseudocode for shared/translation/attrs/S2CombineS1AttrHints

```
// S2CombineS1AttrHints()
// ======================
// Determine resultant Normal memory cacheability and allocation hints from
// combining stage 1 Normal memory attributes and stage 2 cacheability attributes.

MemAttrHints S2CombineS1AttrHints(MemAttrHints s1_attrhints, MemAttrHints s2_attrhints)
    MemAttrHints attrhints;

    if s1_attrhints.attrs == MemAttr_NC || s2_attrhints.attrs == MemAttr_NC then
        attrhints.attrs = MemAttr_NC;
    elsif s1_attrhints.attrs == MemAttr_WT || s2_attrhints.attrs == MemAttr_WT then
        attrhints.attrs = MemAttr_WT;
    else
        attrhints.attrs = MemAttr_WB;

    // Stage 2 does not assign any allocation hints
    // Instead, they are inherited from stage 1
    if attrhints.attrs != MemAttr_NC then
        attrhints.hints     = s1_attrhints.hints;
        attrhints.transient = s1_attrhints.transient;

    return attrhints;
```

Library pseudocode for shared/translation/attrs/S2CombineS1Device

```
// S2CombineS1Device()
// ===================
// Determine resultant Device type from combining output memory attributes
// in stage 1 and Device attributes in stage 2

DeviceType S2CombineS1Device(DeviceType s1_device, DeviceType s2_device)
    if s1_device == DeviceType_nGnRnE || s2_device == DeviceType_nGnRnE then
        return DeviceType_nGnRnE;
    elsif s1_device == DeviceType_nGnRE || s2_device == DeviceType_nGnRE then
        return DeviceType_nGnRE;
    elsif s1_device == DeviceType_nGRE || s2_device == DeviceType_nGRE then
        return DeviceType_nGRE;
    else
        return DeviceType_GRE;
```

Library pseudocode for shared/translation/attrs/S2CombineS1MemAttrs

```
// S2CombineS1MemAttrs()
// =====================
// Combine stage 2 with stage 1 memory attributes

MemoryAttributes S2CombineS1MemAttrs(MemoryAttributes s1_memattrs, MemoryAttributes s2_memattrs,
                                     boolean s2aarch64)
    MemoryAttributes memattrs;

    if s1_memattrs.memtype == MemType_Device && s2_memattrs.memtype == MemType_Device then
        memattrs.memtype = MemType_Device;
        memattrs.device  = S2CombineS1Device(s1_memattrs.device, s2_memattrs.device);
    elsif s1_memattrs.memtype == MemType_Device then    // S2 Normal, S1 Device
        memattrs = s1_memattrs;
    elsif s2_memattrs.memtype == MemType_Device then    // S2 Device, S1 Normal
        memattrs = s2_memattrs;
    else                                                // S2 Normal, S1 Normal
        memattrs.memtype = MemType_Normal;
        memattrs.inner   = S2CombineS1AttrHints(s1_memattrs.inner, s2_memattrs.inner);
        memattrs.outer   = S2CombineS1AttrHints(s1_memattrs.outer, s2_memattrs.outer);

    memattrs.tags = S2MemTagType(memattrs, s1_memattrs.tags);

    if !HaveMTEPermExt() then
        memattrs.notagaccess = FALSE;
    else
        memattrs.notagaccess = (s2_memattrs.notagaccess &&
                               s1_memattrs.tags == MemTag_AllocationTagged);
    memattrs.shareability = S2CombineS1Shareability(s1_memattrs.shareability,
                                                    s2_memattrs.shareability);

    if (memattrs.memtype == MemType_Normal &&
            memattrs.inner.attrs == MemAttr_WB &&
            memattrs.outer.attrs == MemAttr_WB) then
        memattrs.xs = '0';
    elsif s2aarch64 then
        memattrs.xs = s2_memattrs.xs AND s1_memattrs.xs;
    else
        memattrs.xs = s1_memattrs.xs;

    memattrs.shareability = EffectiveShareability(memattrs);
    return memattrs;
```

Library pseudocode for shared/translation/attrs/S2CombineS1Shareability

```
// S2CombineS1Shareability()
// =========================
// Combine stage 2 shareability with stage 1

Shareability S2CombineS1Shareability(Shareability s1_shareability,
                                     Shareability s2_shareability)

    if (s1_shareability == Shareability_OSH ||
            s2_shareability == Shareability_OSH) then
        return Shareability_OSH;
    elsif (s1_shareability == Shareability_ISH ||
            s2_shareability == Shareability_ISH) then
        return Shareability_ISH;
    else
        return Shareability_NSH;
```

Library pseudocode for shared/translation/attrs/S2DecodeCacheability

```
// S2DecodeCacheability()
// ======================
// Determine the stage 2 cacheability for Normal memory

MemAttrHints S2DecodeCacheability(bits(2) attr)
    MemAttrHints s2attr;

    case attr of
        when '01' s2attr.attrs = MemAttr_NC;  // Non-cacheable
        when '10' s2attr.attrs = MemAttr_WT;  // Write-through
        when '11' s2attr.attrs = MemAttr_WB;  // Write-back
        otherwise // Constrained unpredictable
            case ConstrainUnpredictable(Unpredictable_S2RESMEMATTR) of
                when Constraint_NC s2attr.attrs = MemAttr_NC;
                when Constraint_WT s2attr.attrs = MemAttr_WT;
                when Constraint_WB s2attr.attrs = MemAttr_WB;

    // Stage 2 does not assign hints or the transient property
    // They are inherited from stage 1 if the result of the combination allows it
    s2attr.hints     = bits(2) UNKNOWN;
    s2attr.transient = boolean UNKNOWN;

    return s2attr;
```

Library pseudocode for shared/translation/attrs/S2DecodeMemAttrs

```
// S2DecodeMemAttrs()
// ==================
// Decode stage 2 memory attributes

MemoryAttributes S2DecodeMemAttrs(bits(4) attr, bits(2) sh, boolean s2aarch64)
    MemoryAttributes memattrs;

    case attr of
        when '00xx' // Device memory
            memattrs.memtype      = MemType_Device;
            memattrs.device       = DecodeDevice(attr<1:0>);
        when '0100' // Normal, Inner+Outer WB cacheable NoTagAccess memory
            if s2aarch64 && HaveMTEPermExt() then
                memattrs.memtype      = MemType_Normal;
                memattrs.outer        = S2DecodeCacheability('11'); // Write-back
                memattrs.inner        = S2DecodeCacheability('11'); // Write-back
            else
                memattrs.memtype      = MemType_Normal;
                memattrs.outer        = S2DecodeCacheability(attr<3:2>);
                memattrs.inner        = S2DecodeCacheability(attr<1:0>);
        otherwise   // Normal memory
            memattrs.memtype      = MemType_Normal;
            memattrs.outer        = S2DecodeCacheability(attr<3:2>);
            memattrs.inner        = S2DecodeCacheability(attr<1:0>);

    memattrs.shareability = DecodeShareability(sh);

    if s2aarch64 && HaveMTEPermExt() then
        memattrs.notagaccess = attr == '0100';
    else
        memattrs.notagaccess = FALSE;

    return memattrs;
```

Library pseudocode for shared/translation/attrs/S2MemTagType

```
// S2MemTagType()
// ==============
// Determine whether the combined output memory attributes of stage 1 and
// stage 2 indicate tagged memory

MemTagType S2MemTagType(MemoryAttributes s2_memattrs, MemTagType s1_tagtype)

    if !HaveMTE2Ext() then
        return MemTag_Untagged;

    if ((s1_tagtype == MemTag_AllocationTagged)  &&
        (s2_memattrs.memtype == MemType_Normal)  &&
        (s2_memattrs.inner.attrs == MemAttr_WB)  &&
        (s2_memattrs.inner.hints == MemHint_RWA) &&
        (!s2_memattrs.inner.transient)           &&
        (s2_memattrs.outer.attrs == MemAttr_WB)  &&
        (s2_memattrs.outer.hints == MemHint_RWA) &&
        (!s2_memattrs.outer.transient)) then
        return MemTag_AllocationTagged;

    // Return what stage 1 asked for if we can, otherwise Untagged.
    if s1_tagtype != MemTag_AllocationTagged then
        return s1_tagtype;

    return MemTag_Untagged;
```

Library pseudocode for shared/translation/attrs/WalkMemAttrs

```
// WalkMemAttrs()
// ==============
// Retrieve memory attributes of translation table walk

MemoryAttributes WalkMemAttrs(bits(2) sh, bits(2) irgn, bits(2) orgn)
    MemoryAttributes walkmemattrs;

    walkmemattrs.memtype      = MemType_Normal;
    walkmemattrs.shareability = DecodeShareability(sh);
    walkmemattrs.inner        = DecodeSDFAttr(irgn);
    walkmemattrs.outer        = DecodeSDFAttr(orgn);
    walkmemattrs.tags         = MemTag_Untagged;
    if (walkmemattrs.inner.attrs == MemAttr_WB &&
            walkmemattrs.outer.attrs == MemAttr_WB) then
        walkmemattrs.xs = '0';
    else
        walkmemattrs.xs = '1';
    walkmemattrs.notagaccess = FALSE;

    return walkmemattrs;
```

Library pseudocode for shared/translation/faults/AlignmentFault

```
// AlignmentFault()
// ================
// Return a fault record indicating an Alignment fault not due to memory type has occured
// for a specific access

FaultRecord AlignmentFault(AccessDescriptor accdesc)
    FaultRecord fault;

    fault.statuscode  = Fault_Alignment;
    fault.access      = accdesc;
    fault.secondstage = FALSE;
    fault.s2fs1walk   = FALSE;
    fault.write       = !accdesc.read && accdesc.write;
    fault.gpcfs2walk  = FALSE;
    fault.gpcf        = GPCNoFault();

    return fault;
```

Library pseudocode for shared/translation/faults/ExclusiveFault

```
// ExclusiveFault()
// ================
// Return a fault record indicating an Exclusive fault for a specific access

FaultRecord ExclusiveFault(AccessDescriptor accdesc)
    FaultRecord fault;

    fault.statuscode  = Fault_Exclusive;
    fault.access      = accdesc;
    fault.secondstage = FALSE;
    fault.s2fs1walk   = FALSE;
    fault.write       = !accdesc.read && accdesc.write;
    fault.gpcfs2walk  = FALSE;
    fault.gpcf        = GPCNoFault();

    return fault;
```

Library pseudocode for shared/translation/faults/NoFault

```
// NoFault()
// =========
// Return a clear fault record indicating no faults have occured

FaultRecord NoFault()
    FaultRecord fault;

    fault.statuscode  = Fault_None;
    fault.access      = AccessDescriptor UNKNOWN;
    fault.secondstage = FALSE;
    fault.s2fs1walk   = FALSE;
    fault.dirtybit    = FALSE;
    fault.overlay     = FALSE;
    fault.toplevel    = FALSE;
    fault.assuredonly = FALSE;
    fault.s1tagnotdata = FALSE;
    fault.tagaccess   = FALSE;
    fault.gpcfs2walk  = FALSE;
    fault.gpcf        = GPCNoFault();

    return fault;

// NoFault()
// =========
// Return a clear fault record indicating no faults have occured for a specific access

FaultRecord NoFault(AccessDescriptor accdesc)
    FaultRecord fault;

    fault.statuscode  = Fault_None;
    fault.access      = accdesc;
    fault.secondstage = FALSE;
    fault.s2fs1walk   = FALSE;
    fault.dirtybit    = FALSE;
    fault.overlay     = FALSE;
    fault.toplevel    = FALSE;
    fault.assuredonly = FALSE;
    fault.s1tagnotdata = FALSE;
    fault.tagaccess   = FALSE;
    fault.write       = !accdesc.read && accdesc.write;
    fault.gpcfs2walk  = FALSE;
    fault.gpcf        = GPCNoFault();

    return fault;
```

Library pseudocode for shared/translation/gpc/AbovePPS

```
// AbovePPS()
// ==========
// Returns TRUE if an address exceeds the range configured in GPCCR_EL3.PPS.

boolean AbovePPS(bits(56) address)
    pps = DecodePPS();
    if pps >= 56 then
        return FALSE;

    return !IsZero(address<55:pps>);
```

Library pseudocode for shared/translation/gpc/DecodeGPTBlock

```
// DecodeGPTBlock()
// ================
// Validate and decode a GPT Block descriptor

(GPCF, GPTEntry) DecodeGPTBlock(PGSe pgs, bits(64) entry)
    assert entry<3:0> == GPT_Block;
    GPTEntry result;

    if !IsZero(entry<63:8>) then
        return (GPCF_Walk, GPTEntry UNKNOWN);

    if !GPIValid(entry<7:4>) then
        return (GPCF_Walk, GPTEntry UNKNOWN);

    result.gpi   = entry<7:4>;
    result.level = 0;

    // GPT information from a level 0 GPT Block descriptor is permitted
    // to be cached in a TLB as though the Block is a contiguous region
    // of granules each of the size configured in GPCCR_EL3.PGS.
    case pgs of
        when PGS_4KB  result.size = GPTRange_4KB;
        when PGS_16KB result.size = GPTRange_16KB;
        when PGS_64KB result.size = GPTRange_64KB;
        otherwise Unreachable();
    result.contig_size = GPTL0Size();

    return (GPCF_None, result);
```

Library pseudocode for shared/translation/gpc/DecodeGPTContiguous

```
// DecodeGPTContiguous()
// =====================
// Validate and decode a GPT Contiguous descriptor

(GPCF, GPTEntry) DecodeGPTContiguous(PGSe pgs, bits(64) entry)
    assert entry<3:0> == GPT_Contig;
    GPTEntry result;

    if !IsZero(entry<63:10>) then
        return (GPCF_Walk, result);

    result.gpi = entry<7:4>;
    if !GPIValid(result.gpi) then
        return (GPCF_Walk, result);

    case pgs of
        when PGS_4KB  result.size = GPTRange_4KB;
        when PGS_16KB result.size = GPTRange_16KB;
        when PGS_64KB result.size = GPTRange_64KB;
        otherwise Unreachable();

    case entry<9:8> of
        when '01' result.contig_size = GPTRange_2MB;
        when '10' result.contig_size = GPTRange_32MB;
        when '11' result.contig_size = GPTRange_512MB;
        otherwise return (GPCF_Walk, GPTEntry UNKNOWN);

    result.level = 1;

    return (GPCF_None, result);
```

Library pseudocode for shared/translation/gpc/DecodeGPTGranules

```
// DecodeGPTGranules()
// ===================
// Validate and decode a GPT Granules descriptor

(GPCF, GPTEntry) DecodeGPTGranules(PGSe pgs, integer index, bits(64) entry)
    GPTEntry result;

    for i = 0 to 15
        if !GPIValid(entry<i*4 +:4>) then
            return (GPCF_Walk, result);

    result.gpi = entry<index*4 +:4>;

    case pgs of
        when PGS_4KB  result.size = GPTRange_4KB;
        when PGS_16KB result.size = GPTRange_16KB;
        when PGS_64KB result.size = GPTRange_64KB;
        otherwise Unreachable();

    result.contig_size = result.size; // No contiguity
    result.level = 1;

    return (GPCF_None, result);
```

Library pseudocode for shared/translation/gpc/DecodeGPTTable

```
// DecodeGPTTable()
// ================
// Validate and decode a GPT Table descriptor

(GPCF, GPTTable) DecodeGPTTable(PGSe pgs, bits(64) entry)
    assert entry<3:0> == GPT_Table;
    GPTTable result;

    if !IsZero(entry<63:52,11:4>) then
        return (GPCF_Walk, GPTTable UNKNOWN);

    l0sz = GPTL0Size();
    integer p;
    case pgs of
        when PGS_4KB  p = 12;
        when PGS_16KB p = 14;
        when PGS_64KB p = 16;
        otherwise Unreachable();

    if !IsZero(entry<(l0sz-p)-2:12>) then
        return (GPCF_Walk, GPTTable UNKNOWN);

    case pgs of
        when PGS_4KB  result.address = entry<55:17>:Zeros(17);
        when PGS_16KB result.address = entry<55:15>:Zeros(15);
        when PGS_64KB result.address = entry<55:13>:Zeros(13);
        otherwise Unreachable();

    // The address must be within the range covered by the GPT
    if AbovePPS(result.address) then
        return (GPCF_AddressSize, GPTTable UNKNOWN);

    return (GPCF_None, result);
```

Library pseudocode for shared/translation/gpc/DecodePGS

```
// DecodePGS()
// ===========

PGSe DecodePGS(bits(2) pgs)
    case pgs of
        when '00' return PGS_4KB;
        when '10' return PGS_16KB;
        when '01' return PGS_64KB;
        otherwise Unreachable();
```

Library pseudocode for shared/translation/gpc/DecodePPS

```
// DecodePPS()
// ===========
// Size of region protected by the GPT, in bits.

integer DecodePPS()
    case GPCCR_EL3.PPS of
        when '000' return 32;
        when '001' return 36;
        when '010' return 40;
        when '011' return 42;
        when '100' return 44;
        when '101' return 48;
        when '110' return 52;
        otherwise Unreachable();
```

Library pseudocode for shared/translation/gpc/GPCFault

```
// GPCFault()
// ==========
// Constructs and returns a GPCF

GPCFRecord GPCFault(GPCF gpf, integer level)
    GPCFRecord fault;
    fault.gpf   = gpf;
    fault.level = level;
    return fault;
```

Library pseudocode for shared/translation/gpc/GPCNoFault

```
// GPCNoFault()
// ============
// Returns the default properties of a GPCF that does not represent a fault

GPCFRecord GPCNoFault()
    GPCFRecord result;
    result.gpf = GPCF_None;
    return result;
```

Library pseudocode for shared/translation/gpc/GPCRegistersConsistent

```
// GPCRegistersConsistent()
// ========================
// Returns whether the GPT registers are configured correctly.
// This returns false if any fields select a Reserved value.

boolean GPCRegistersConsistent()

    // Check for Reserved register values
    if GPCCR_EL3.PPS == '111' || DecodePPS() > AArch64.PAMax() then
        return FALSE;
    if GPCCR_EL3.PGS == '11' then
        return FALSE;
    if GPCCR_EL3.SH == '01' then
        return FALSE;

    // Inner and Outer Non-cacheable requires Outer Shareable
    if GPCCR_EL3.<ORGN, IRGN> == '0000' && GPCCR_EL3.SH != '10' then
        return FALSE;

    return TRUE;
```

Library pseudocode for shared/translation/gpc/GPICheck

```
// GPICheck()
// ==========
// Returns whether an access to a given physical address space is permitted
// given the configured GPI value.
// paspace: Physical address space of the access
// gpi: Value read from GPT for the access

boolean GPICheck(PASpace paspace, bits(4) gpi)

    case gpi of
        when GPT_NoAccess  return FALSE;
        when GPT_Secure    assert HaveSecureEL2Ext();return paspace == PAS_Secure;
        when GPT_NonSecure return paspace == PAS_NonSecure;
        when GPT_Root      return paspace == PAS_Root;
        when GPT_Realm     return paspace == PAS_Realm;
        when GPT_Any       return TRUE;
        otherwise          Unreachable();
```

Library pseudocode for shared/translation/gpc/GPIIndex

```
// GPIIndex()
// ==========

integer GPIIndex(bits(56) pa)
    case DecodePGS(GPCCR_EL3.PGS) of
        when PGS_4KB  return UInt(pa<15:12>);
        when PGS_16KB return UInt(pa<17:14>);
        when PGS_64KB return UInt(pa<19:16>);
        otherwise Unreachable();
```

Library pseudocode for shared/translation/gpc/GPIValid

```
// GPIValid()
// ==========
// Returns whether a given value is a valid encoding for a GPI value

boolean GPIValid(bits(4) gpi)
    if gpi == GPT_Secure then
        return HaveSecureEL2Ext();

    return gpi IN {GPT_NoAccess,
                   GPT_NonSecure,
                   GPT_Root,
                   GPT_Realm,
                   GPT_Any};
```

Library pseudocode for shared/translation/gpc/GPTL0Size

```
// GPTL0Size()
// ===========
// Returns number of bits covered by a level 0 GPT entry

integer GPTL0Size()
    case GPCCR_EL3.L0GPTSZ of
        when '0000' return GPTRange_1GB;
        when '0100' return GPTRange_16GB;
        when '0110' return GPTRange_64GB;
        when '1001' return GPTRange_512GB;
        otherwise Unreachable();
    return 30;
```

Library pseudocode for shared/translation/gpc/GPTLevel0Index

```
// GPTLevel0Index()
// ================
// Compute the level 0 index based on input PA.

integer GPTLevel0Index(bits(56) pa)
    // Input address and index bounds
    pps = DecodePPS();
    l0sz = GPTL0Size();
    if pps <= l0sz then
        return 0;

    return UInt(pa<pps-1:l0sz>);
```

Library pseudocode for shared/translation/gpc/GPTLevel1Index

```
// GPTLevel1Index()
// ================
// Compute the level 1 index based on input PA.

integer GPTLevel1Index(bits(56) pa)
    // Input address and index bounds
    l0sz = GPTL0Size();
    case DecodePGS(GPCCR_EL3.PGS) of
        when PGS_4KB  return UInt(pa<l0sz-1:16>);
        when PGS_16KB return UInt(pa<l0sz-1:18>);
        when PGS_64KB return UInt(pa<l0sz-1:20>);
        otherwise Unreachable();
```

Library pseudocode for shared/translation/gpc/GPTWalk

```
// GPTWalk()
// =========
// Get the GPT entry for a given physical address, pa

(GPCFRecord, GPTEntry) GPTWalk(bits(56) pa, AccessDescriptor accdesc)

    // GPT base address
    bits(56) base;
    pgs = DecodePGS(GPCCR_EL3.PGS);

    // The level 0 GPT base address is aligned to the greater of:
    // * the size of the level 0 GPT, determined by GPCCR_EL3.{PPS, L0GPTSZ}.
    // * 4KB
    base = ZeroExtend(GPTBR_EL3.BADDR:Zeros(12), 56);
    pps = DecodePPS();
    l0sz = GPTL0Size();
    integer alignment = Max((pps - l0sz) + 3, 12);
    base = base AND NOT ZeroExtend(Ones(alignment), 56);

    AccessDescriptor gptaccdesc = CreateAccDescGPTW(accdesc);

    // Access attributes and address for GPT fetches
    AddressDescriptor gptaddrdesc;
    gptaddrdesc.memattrs = WalkMemAttrs(GPCCR_EL3.SH, GPCCR_EL3.ORGN, GPCCR_EL3.IRGN);
    gptaddrdesc.fault    = NoFault(gptaccdesc);

    // Address of level 0 GPT entry
    gptaddrdesc.paddress.paspace = PAS_Root;
    gptaddrdesc.paddress.address = base + GPTLevel0Index(pa) * 8;

    // Fetch L0GPT entry
    bits(64) level_0_entry;
    PhysMemRetStatus memstatus;
    (memstatus, level_0_entry) = PhysMemRead(gptaddrdesc, 8, gptaccdesc);
    if IsFault(memstatus) then
        return (GPCFault(GPCF_EABT, 0), GPTEntry UNKNOWN);

    GPTEntry result;
    GPTTable table;
    GPCF gpf;
    case level_0_entry<3:0> of
        when GPT_Block
            // Decode the GPI value and return that
            (gpf, result) = DecodeGPTBlock(pgs, level_0_entry);
            result.pa = pa;
            return (GPCFault(gpf, 0), result);
        when GPT_Table
            // Decode the table entry and continue walking
            (gpf, table) = DecodeGPTTable(pgs, level_0_entry);
            if gpf != GPCF_None then
                return (GPCFault(gpf, 0), GPTEntry UNKNOWN);
        otherwise
            // GPF - invalid encoding
            return (GPCFault(GPCF_Walk, 0), GPTEntry UNKNOWN);

    // Must be a GPT Table entry
    assert level_0_entry<3:0> == GPT_Table;

    // Address of level 1 GPT entry
    offset = GPTLevel1Index(pa) * 8;
    gptaddrdesc.paddress.address = table.address + offset;

    // Fetch L1GPT entry
    bits(64) level_1_entry;
    (memstatus, level_1_entry) = PhysMemRead(gptaddrdesc, 8, gptaccdesc);
    if IsFault(memstatus) then
        return (GPCFault(GPCF_EABT, 1), GPTEntry UNKNOWN);

    case level_1_entry<3:0> of
        when GPT_Contig
```

```
            (gpf, result) = DecodeGPTContiguous(pgs, level_1_entry);
        otherwise
            gpi_index = GPIIndex(pa);
            (gpf, result) = DecodeGPTGranules(pgs, gpi_index, level_1_entry);

    if gpf != GPCF_None then
        return (GPCFault(gpf, 1), GPTEntry UNKNOWN);

    result.pa = pa;
    return (GPCNoFault(), result);
```

Library pseudocode for shared/translation/gpc/GranuleProtectionCheck

```
// GranuleProtectionCheck()
// ========================
// Returns whether a given access is permitted, according to the
// granule protection check.
// addrdesc and accdesc describe the access to be checked.

GPCFRecord GranuleProtectionCheck(AddressDescriptor addrdesc, AccessDescriptor accdesc)

    assert HaveRME();
    // The address to be checked
    address = addrdesc.paddress;

    // Bypass mode - all accesses pass
    if GPCCR_EL3.GPC == '0' then
        return GPCNoFault();

    // Configuration consistency check
    if !GPCRegistersConsistent() then
        return GPCFault(GPCF_Walk, 0);

    // Input address size check
    if AbovePPS(address.address) then
        if address.paspace == PAS_NonSecure then
            return GPCNoFault();
        else
            return GPCFault(GPCF_Fail, 0);

    // GPT base address size check
    bits(56) gpt_base = ZeroExtend(GPTBR_EL3.BADDR:Zeros(12), 56);
    if AbovePPS(gpt_base) then
        return GPCFault(GPCF_AddressSize, 0);

    // GPT lookup
    (gpcf, gpt_entry) = GPTWalk(address.address, accdesc);
    if gpcf.gpf != GPCF_None then
        return gpcf;

    // Check input physical address space against GPI
    permitted = GPICheck(address.paspace, gpt_entry.gpi);

    if !permitted then
        gpcf = GPCFault(GPCF_Fail, gpt_entry.level);
        return gpcf;

    // Check passed

    return GPCNoFault();
```

Library pseudocode for shared/translation/gpc/PGS

```
// PGS
// ===
// Physical granule size

enumeration PGSe {
    PGS_4KB,
    PGS_16KB,
    PGS_64KB
};
```

Library pseudocode for shared/translation/gpc/Table

```
constant bits(4) GPT_NoAccess  = '0000';
constant bits(4) GPT_Table     = '0011';
constant bits(4) GPT_Block     = '0001';
constant bits(4) GPT_Contig    = '0001';
constant bits(4) GPT_Secure    = '1000';
constant bits(4) GPT_NonSecure = '1001';
constant bits(4) GPT_Root      = '1010';
constant bits(4) GPT_Realm     = '1011';
constant bits(4) GPT_Any       = '1111';
constant integer GPTRange_4KB   = 12;
constant integer GPTRange_16KB  = 14;
constant integer GPTRange_64KB  = 16;
constant integer GPTRange_2MB   = 21;
constant integer GPTRange_32MB  = 25;
constant integer GPTRange_512MB = 29;
constant integer GPTRange_1GB   = 30;
constant integer GPTRange_16GB  = 34;
constant integer GPTRange_64GB  = 36;
constant integer GPTRange_512GB = 39;

type GPTTable is (
    bits(56) address        // Base address of next table
)

type GPTEntry is (
    bits(4)  gpi,           // GPI value for this region
    integer  size,          // Region size
    integer  contig_size,   // Contiguous region size
    integer  level,         // Level of GPT lookup
    bits(56) pa             // PA uniquely identifying the GPT entry
)
```

Library pseudocode for shared/translation/translation/S1TranslationRegime

```
// S1TranslationRegime()
// =====================
// Stage 1 translation regime for the given Exception level

bits(2) S1TranslationRegime(bits(2) el)
    if el != EL0 then
        return el;
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) && SCR.NS == '0' then
        return EL3;
    elsif HaveVirtHostExt() && ELIsInHost(el) then
        return EL2;
    else
        return EL1;

// S1TranslationRegime()
// =====================
// Returns the Exception level controlling the current Stage 1 translation regime. For the most
// part this is unused in code because the System register accessors (SCTLR[], etc.) implicitly
// return the correct value.

bits(2) S1TranslationRegime()
    return S1TranslationRegime(PSTATE.EL);
```

Library pseudocode for shared/translation/vmsa/AddressDescriptor

```
constant integer FINAL_LEVEL = 3;

// AddressDescriptor
// =================
// Descriptor used to access the underlying memory array.

type AddressDescriptor is (
    FaultRecord      fault,      // fault.statuscode indicates whether the address is valid
    MemoryAttributes memattrs,
    FullAddress      paddress,
    boolean          s1assured,  // Stage 1 Assured Translation Property
    boolean          s2fs1mro,   // Stage 2 MRO permission for Satge 1
    bits(16)         mecid,      // FEAT_MEC: Memory Encryption Context ID
    bits(64)         vaddress
)
```

Library pseudocode for shared/translation/vmsa/ContiguousSize

```
// ContiguousSize()
// ================
// Return the number of entries log 2 marking a contiguous output range

integer ContiguousSize(bit d128, TGx tgx, integer level)
    if d128 == '1' then
        case tgx of
            when TGx_4KB
                assert level IN {1, 2, 3};
                return if level == 1 then 2 else 4;
            when TGx_16KB
                assert level IN {1, 2, 3};
                if level == 1 then
                    return 2;
                elsif level == 2 then
                    return 4;
                else
                    return 6;
            when TGx_64KB
                assert level IN {2, 3};
                return if level == 2 then 6 else 4;
    else
        case tgx of
            when TGx_4KB
                assert level IN {1, 2, 3};
                return 4;
            when TGx_16KB
                assert level IN {2, 3};
                return if level == 2 then 5 else 7;
            when TGx_64KB
                assert level IN {2, 3};
                return 5;
```

Library pseudocode for shared/translation/vmsa/CreateAddressDescriptor

```
// CreateAddressDescriptor()
// =========================
// Set internal members for address descriptor type to valid values

AddressDescriptor CreateAddressDescriptor(bits(64) va, FullAddress pa,
                                          MemoryAttributes memattrs)
    AddressDescriptor addrdesc;

    addrdesc.paddress = pa;
    addrdesc.vaddress = va;
    addrdesc.memattrs = memattrs;
    addrdesc.fault    = NoFault();
    addrdesc.s1assured = FALSE;

    return addrdesc;
```

Library pseudocode for shared/translation/vmsa/CreateFaultyAddressDescriptor

```
// CreateFaultyAddressDescriptor()
// ===============================
// Set internal members for address descriptor type with values indicating error

AddressDescriptor CreateFaultyAddressDescriptor(bits(64) va, FaultRecord fault)
    AddressDescriptor addrdesc;

    addrdesc.vaddress = va;
    addrdesc.fault    = fault;

    return addrdesc;
```

Library pseudocode for shared/translation/vmsa/DecodePASpace

```
// DecodePASpace()
// ===============
// Decode the target PA Space

PASpace DecodePASpace (bit nse, bit ns)
    case nse:ns of
        when '00'   return PAS_Secure;
        when '01'   return PAS_NonSecure;
        when '10'   return PAS_Root;
        when '11'   return PAS_Realm;
```

Library pseudocode for shared/translation/vmsa/DescriptorType

```
// DescriptorType
// ==============
// Translation table descriptor formats

enumeration DescriptorType {
    DescriptorType_Table,
    DescriptorType_Leaf,
    DescriptorType_Invalid
};
```

Library pseudocode for shared/translation/vmsa/Domains

```
constant bits(2) Domain_NoAccess = '00';
constant bits(2) Domain_Client   = '01';
constant bits(2) Domain_Manager  = '11';
```

Library pseudocode for shared/translation/vmsa/FetchDescriptor

```
// FetchDescriptor()
// =================
// Fetch a translation table descriptor

(FaultRecord, bits(N)) FetchDescriptor(bit ee, AddressDescriptor walkaddress,
                                       AccessDescriptor walkaccess, FaultRecord fault_in,
                                       integer N)
    // 32-bit descriptors for AArch32 Short-descriptor format
    // 64-bit descriptors for AArch64 or AArch32 Long-descriptor format
    // 128-bit descriptors for AArch64 when FEAT_D128 is set and {V}TCR_ELx.d128 is set
    assert N == 32 || N == 64 || N == 128;
    bits(N) descriptor;
    FaultRecord fault = fault_in;

    if HaveRME() then
        fault.gpcf = GranuleProtectionCheck(walkaddress, walkaccess);
        if fault.gpcf.gpf != GPCF_None then
            fault.statuscode = Fault_GPCFOnWalk;
            fault.paddress   = walkaddress.paddress;
            fault.gpcfs2walk = fault.secondstage;
            return (fault, bits(N) UNKNOWN);

    PhysMemRetStatus memstatus;
    (memstatus, descriptor) = PhysMemRead(walkaddress, N DIV 8, walkaccess);
    if IsFault(memstatus) then
        boolean iswrite = FALSE;
        fault = HandleExternalTTWAbort(memstatus, iswrite, walkaddress,
                                       walkaccess, N DIV 8, fault);
        if IsFault(fault.statuscode) then
            return (fault, bits(N) UNKNOWN);

    if ee == '1' then
        descriptor = BigEndianReverse(descriptor);

    return (fault, descriptor);
```

Library pseudocode for shared/translation/vmsa/HasUnprivileged

```
// HasUnprivileged()
// =================
// Returns whether a translation regime serves EL0 as well as a higher EL

boolean HasUnprivileged(Regime regime)
    return (regime IN {
        Regime_EL20,
        Regime_EL30,
        Regime_EL10
    });
```

Library pseudocode for shared/translation/vmsa/Regime

```
// Regime
// ======
// Translation regimes

enumeration Regime {
    Regime_EL3,            // EL3
    Regime_EL30,           // EL3&0 (PL1&0 when EL3 is AArch32)
    Regime_EL2,            // EL2
    Regime_EL20,           // EL2&0
    Regime_EL10            // EL1&0
};
```

Library pseudocode for shared/translation/vmsa/RegimeUsingAArch32

```
// RegimeUsingAArch32()
// ====================
// Determine if the EL controlling the regime executes in AArch32 state

boolean RegimeUsingAArch32(Regime regime)
    case regime of
        when Regime_EL10 return ELUsingAArch32(EL1);
        when Regime_EL30 return TRUE;
        when Regime_EL20 return FALSE;
        when Regime_EL2  return ELUsingAArch32(EL2);
        when Regime_EL3  return FALSE;
```

Library pseudocode for shared/translation/vmsa/S1TTWParams

```
// S1TTWParams
// ===========
// Register fields corresponding to stage 1 translation
// For A32-VMSA, if noted, they correspond to A32-LPAE (Long descriptor format)

type S1TTWParams is (
// A64-VMSA exclusive parameters
    bit         ha,         // TCR_ELx.HA
    bit         hd,         // TCR_ELx.HD
    bit         tbi,        // TCR_ELx.TBI{x}
    bit         tbid,       // TCR_ELx.TBID{x}
    bit         nfd,        // TCR_EL1.NFDx or TCR_EL2.NFDx when HCR_EL2.E2H == '1'
    bit         e0pd,       // TCR_EL1.E0PDx or TCR_EL2.E0PDx when HCR_EL2.E2H == '1'
    bit         d128,       // TCR_ELx.D128
    bit         aie,         // (TCR2_ELx/TCR_EL3).AIE
    MAIRType    mair2,       // MAIR2_ELx
    bit         ds,         // TCR_ELx.DS
    bits(3)     ps,         // TCR_ELx.{I}PS
    bits(6)     txsz,       // TCR_ELx.TxSZ
    bit         epan,       // SCTLR_EL1.EPAN or SCTLR_EL2.EPAN when HCR_EL2.E2H == '1'
    bit         dct,        // HCR_EL2.DCT
    bit         nv1,        // HCR_EL2.NV1
    bit         cmow,       // SCTLR_EL1.CMOW or SCTLR_EL2.CMOW when HCR_EL2.E2H == '1'
    bit         pnch,       // TCR{2}_ELx.PnCH
    bit         disch,      // TCR{2}_ELx.DisCH
    bit         haft,       // TCR{2}_ELx.HAFT
    bit         mtx,        // TCR_ELx.MTX{y}
    bits(2)     skl,        // TCR_ELx.SKL
    bit         pie,        // TCR2_ELx.PIE or TCR_EL3.PIE
    S1PIRType   pir,        // PIR_ELx
    S1PIRType   pire0,      // PIRE0_EL1 or PIRE0_EL2 when HCR_EL2.E2H == '1'
    bit         emec,       // SCTLR2_EL2.EMEC or SCTLR2_EL3.EMEC
    bit         amec,       // TCR2_EL2.AMEC0 or TCR2_EL2.AMEC1 when HCR_EL2.E2H == '1'

// A32-VMSA exclusive parameters
    bits(3)     t0sz,       // TTBCR.T0SZ
    bits(3)     t1sz,       // TTBCR.T1SZ
    bit         uwxn,       // SCTLR.UWXN

// Parameters common to both A64-VMSA & A32-VMSA (A64/A32)
    TGx         tgx,        // TCR_ELx.TGx      / Always TGx_4KB
    bits(2)     irgn,       // TCR_ELx.IRGNx    / TTBCR.IRGNx or HTCR.IRGN0
    bits(2)     orgn,       // TCR_ELx.ORGNx    / TTBCR.ORGNx or HTCR.ORGN0
    bits(2)     sh,         // TCR_ELx.SHx      / TTBCR.SHx or HTCR.SH0
    bit         hpd,        // TCR_ELx.HPD{x}   / TTBCR2.HPDx or HTCR.HPD
    bit         ee,         // SCTLR_ELx.EE     / SCTLR.EE or HSCTLR.EE
    bit         wxn,        // SCTLR_ELx.WXN    / SCTLR.WXN or HSCTLR.WXN
    bit         ntlsmd,     // SCTLR_ELx.nTLSMD / SCTLR.nTLSMD or HSCTLR.nTLSMD
    bit         dc,         // HCR_EL2.DC       / HCR.DC
    bit         sif,        // SCR_EL3.SIF      / SCR.SIF
    MAIRType    mair        // MAIR_ELx         / MAIR1:MAIR0 or HMAIR1:HMAIR0
)
```

Library pseudocode for shared/translation/vmsa/S2TTWParams

```
// S2TTWParams
// ===========
// Register fields corresponding to stage 2 translation.

type S2TTWParams is (
// A64-VMSA exclusive parameters
    bit         ha,         // VTCR_EL2.HA
    bit         hd,         // VTCR_EL2.HD
    bit         sl2,        // V{S}TCR_EL2.SL2
    bit         ds,         // VTCR_EL2.DS
    bit         d128,       // VTCR_ELx.D128
    bit         sw,         // VSTCR_EL2.SW
    bit         nsw,        // VTCR_EL2.NSW
    bit         sa,         // VSTCR_EL2.SA
    bit         nsa,        // VTCR_EL2.NSA
    bits(3)     ps,         // VTCR_EL2.PS
    bits(6)     txsz,       // V{S}TCR_EL2.T0SZ
    bit         fwb,        // HCR_EL2.PTW
    bit         cmow,       // HCRX_EL2.CMOW
    bits(2)     skl,        // VTCR_EL2.SKL
    bit         s2pie,      // VTCR_EL2.S2PIE
    S2PIRType   s2pir,      // S2PIR_EL2
    bit         tl0,        // VTCR_EL2.TL0
    bit         tl1,        // VTCR_EL2.TL1
    bit         assuredonly,// VTCR_EL2.AssuredOnly
    bit         haft,       // VTCR_EL2.HAFT
    bit         emec,       // SCTLR2_EL2.EMEC

// A32-VMSA exclusive parameters
    bit         s,          // VTCR.S
    bits(4)     t0sz,       // VTCR.T0SZ

// Parameters common to both A64-VMSA & A32-VMSA if implemented (A64/A32)
    TGx         tgx,        // V{S}TCR_EL2.TG0  / Always TGx_4KB
    bits(2)     sl0,        // V{S}TCR_EL2.SL0  / VTCR.SL0
    bits(2)     irgn,       // VTCR_EL2.IRGN0   / VTCR.IRGN0
    bits(2)     orgn,       // VTCR_EL2.ORGN0   / VTCR.ORGN0
    bits(2)     sh,         // VTCR_EL2.SH0     / VTCR.SH0
    bit         ee,         // SCTLR_EL2.EE     / HSCTLR.EE
    bit         ptw,        // HCR_EL2.PTW      / HCR.PTW
    bit         vm          // HCR_EL2.VM       / HCR.VM
)
```

Library pseudocode for shared/translation/vmsa/SDFType

```
// SDFType
// =======
// Short-descriptor format type

enumeration SDFType {
    SDFType_Table,
    SDFType_Invalid,
    SDFType_Supersection,
    SDFType_Section,
    SDFType_LargePage,
    SDFType_SmallPage
};
```

Library pseudocode for shared/translation/vmsa/SecurityStateForRegime

```
// SecurityStateForRegime()
// ========================
// Return the Security State of the given translation regime

SecurityState SecurityStateForRegime(Regime regime)
    case regime of
        when Regime_EL3     return SecurityStateAtEL(EL3);
        when Regime_EL30    return SS_Secure; // A32 EL3 is always Secure
        when Regime_EL2     return SecurityStateAtEL(EL2);
        when Regime_EL20    return SecurityStateAtEL(EL2);
        when Regime_EL10    return SecurityStateAtEL(EL1);
```

Library pseudocode for shared/translation/vmsa/StageOA

```
// StageOA()
// =========
// Given the final walk state (a page or block descriptor), map the untranslated
// input address bits to the output address

FullAddress StageOA(bits(64) ia, bit d128, TGx tgx, TTWState walkstate)
    // Output Address
    FullAddress oa;
    integer csize;

    tsize = TranslationSize(d128, tgx, walkstate.level);
    if walkstate.contiguous == '1' then
        csize = ContiguousSize(d128, tgx, walkstate.level);
    else
        csize = 0;

    ia_msb = tsize + csize;
    oa.paspace = walkstate.baseaddress.paspace;
    oa.address = walkstate.baseaddress.address<55:ia_msb>:ia<ia_msb-1:0>;

    return oa;
```

Library pseudocode for shared/translation/vmsa/TGx

```
// TGx
// ===
// Translation granules sizes

enumeration TGx {
    TGx_4KB,
    TGx_16KB,
    TGx_64KB
};
```

Library pseudocode for shared/translation/vmsa/TGxGranuleBits

```
// TGxGranuleBits()
// ================
// Retrieve the address size, in bits, of a granule

integer TGxGranuleBits(TGx tgx)
    case tgx of
        when TGx_4KB  return 12;
        when TGx_16KB return 14;
        when TGx_64KB return 16;
```

Library pseudocode for shared/translation/vmsa/TLBContext

```
// TLBContext
// ==========
// Translation context compared on TLB lookups and invalidations, promoting a TLB hit on match

type TLBContext is (
    SecurityState ss,
    Regime        regime,
    bits(16)      vmid,
    bits(16)      asid,
    bit           nG,
    PASpace       ipaspace, // Used in stage 2 lookups & invalidations only
    boolean       includes_s1,
    boolean       includes_s2,
    boolean       includes_gpt,
    bits(64)      ia,       // Input Address
    TGx           tg,
    bit           cnp,
    integer       level,    // Assist TLBI level hints (FEAT_TTL)
    boolean       isd128,
    bit           xs        // XS attribute (FEAT_XS)
)
```

Library pseudocode for shared/translation/vmsa/TLBRecord

```
// TLBRecord
// =========
// Translation output as a TLB payload

type TLBRecord is (
    TLBContext  context,
    TTWState    walkstate,
    integer     blocksize,    // Number of bits directly mapped from IA to OA
    integer     contigsize,   // Number of entries log 2 marking a contiguous output range
    bits(128)   s1descriptor, // Stage 1 leaf descriptor in memory (valid if the TLB caches stage 1)
    bits(128)   s2descriptor  // Stage 2 leaf descriptor in memory (valid if the TLB caches stage 2)
)
```

Library pseudocode for shared/translation/vmsa/TTWState

```
// TTWState
// ========
// Translation table walk state

type TTWState is (
    boolean             istable,
    integer             level,
    FullAddress         baseaddress,
    bit                 contiguous,
    boolean             s1assured,      // Stage 1 Assured Translation Property
    bit                 s2assuredonly,  // Stage 2 AssuredOnly attribute
    bit                 disch,          // Stage 1 Disable Contiguous Hint
    bit                 nG,
    bit                 guardedpage,
    SDFType             sdftype,    // AArch32 Short-descriptor format walk only
    bits(4)             domain,     // AArch32 Short-descriptor format walk only
    MemoryAttributes    memattrs,
    Permissions         permissions
)
```

Library pseudocode for shared/translation/vmsa/TranslationRegime

```
// TranslationRegime()
// ===================
// Select the translation regime given the target EL and PE state

Regime TranslationRegime(bits(2) el)
    if el == EL3 then
        return if ELUsingAArch32(EL3) then Regime_EL30 else Regime_EL3;
    elsif el == EL2 then
        return if ELIsInHost(EL2) then Regime_EL20 else Regime_EL2;
    elsif el == EL1 then
        return Regime_EL10;
    elsif el == EL0 then
        if CurrentSecurityState() == SS_Secure && ELUsingAArch32(EL3) then
            return Regime_EL30;
        elsif ELIsInHost(EL0) then
            return Regime_EL20;
        else
            return Regime_EL10;
    else
        Unreachable();
```

Library pseudocode for shared/translation/vmsa/TranslationSize

```
// TranslationSize()
// =================
// Compute the number of bits directly mapped from the input address
// to the output address

integer TranslationSize(bit d128, TGx tgx, integer level)
    granulebits = TGxGranuleBits(tgx);
    descsizelog2 = if d128 == '1' then 4 else 3;
    blockbits   = (FINAL_LEVEL - level) * (granulebits - descsizelog2);

    return granulebits + blockbits;
```

Library pseudocode for shared/translation/vmsa/UseASID

```
// UseASID()
// =========
// Determine whether the translation context for the access requires ASID or is a global entry

boolean UseASID(TLBContext access)
    return HasUnprivileged(access.regime);
```

Library pseudocode for shared/translation/vmsa/UseVMID

```
// UseVMID()
// =========
// Determine whether the translation context for the access requires VMID to match a TLB entry

boolean UseVMID(TLBContext access)
    return access.regime == Regime_EL10 && EL2Enabled();
```

Internal version only: isa v01_31, pseudocode v2023-03_rel, sve v2023-03_rc3b ; Build timestamp: 2023-03-31T10:19