C6.1 About the A64 base instructions

Alphabetical list of A64 base instructions gives full descriptions of the A64 instructions that are in the following instruction groups:

- Branch, Exception generation, and System instructions.
- Loads and stores associated with the general-purpose registers.
- Data processing (immediate).
- Data processing (register).

A64 instruction set encoding provides an overview of the instruction encodings as well as of the instruction classes within their functional groups.

The rest of this section is general description of the base instructions. It contains the following subsections:

- *Register size*.
- *Use of the PC*.
- *Use of the stack pointer*.
- *Condition flags and related instructions*.

C6.1.1 Register size

Most data processing, comparison, and conversion instructions that use the general-purpose registers as the source or destination operand have two instruction variants that operate on either a 32-bit or a 64-bit value.

Where a 32-bit instruction form is selected, the following holds:

- The upper 32 bits of the source registers are ignored.
- The upper 32 bits of the destination register are set to zero.
- Right shifts and right rotates inject at bit[31], not at bit[63].
- The Condition flags, where set by the instruction, are computed from the lower 32 bits.

This distinction applies even when the results of a 32-bit instruction form are indistinguishable from the lower 32 bits computed by the equivalent 64-bit instruction form. For example, a 32-bit bitwise `ORR` could be performed using a 64-bit `ORR` and simply ignoring the top 32 bits of the result. However, the A64 instruction set includes separate 32-bit and 64-bit forms of the `ORR` instruction.

As well as distinct sign-extend or zero-extend instructions, the A64 instruction set also provides the ability to extend and shift the final source register of an `ADD`, `SUB`, `ADDS`, `or SUBS` instruction and the index register of a load/store instruction. This enables array index calculations involving a 64-bit array pointer and a 32-bit array index to be implemented efficiently.

The assembly language notation enables the distinct identification of registers holding 32-bit values and registers holding 64-bit values. See *Register names* and *Register indexed addressing*.

C6.1.2 Use of the PC

A64 instructions have limited access to the PC. The only instructions that can read the PC are those that generate a PC relative address:

- ADR and ADRP.
- The Load register (literal) instruction class.
- Direct branches that use an immediate offset.
- The unconditional branch with link instructions, BL and BLR, that use the PC to create the return link address.

Only explicit control flow instructions can modify the PC:

- Conditional and unconditional branch and return instructions.
- Exception generation and exception return instructions.

For more details of instructions that can modify the PC, see *Branches, Exception generating, and System instructions*.

C6.1.3 Use of the stack pointer

A64 instructions can use the stack pointer only in a limited number of cases:

- Load/store instructions use the current stack pointer as the base address:
  - When stack alignment checking is enabled by system software and the base register is SP, the current stack pointer must be initially quadword aligned, That is, it must be aligned to 16 bytes. Misalignment generates an SP alignment fault. See *SP alignment checking* for more information.
- Add and subtract data processing instructions in their immediate and extended register forms, use the current stack pointer as a source register or the destination register or both.
- Logical data processing instructions in their immediate form use the current stack pointer as the destination register.

C6.1.4 Condition flags and related instructions

The A64 base instructions that use the Condition flags as an input are:

- Conditional branch. The conditional branch instruction is `B.cond`.
- Add or subtract with carry. These instruction types include instructions to perform multi-precision arithmetic and calculate checksums. The add or subtract with carry instructions are `ADC`, `ADCS`, `SBC`, and `SBCS`, or an architectural alias for these instructions.
- Conditional select with increment, negate, or invert. This instruction type conditionally selects between one source register and a second, incremented, negated, inverted, or unmodified source register. The conditional select with increment, negate, or invert instructions are `CSINC`, `CSINV`, and `CSNEG`.

  These instructions also implement:
  - Conditional select or move. The Condition flags select one of two source registers as the destination register. Short conditional sequences can be replaced by unconditional instructions followed by a conditional select, `CSEL`.
  - Conditional set. Conditionally selects between 0 and 1, or 0 and -1. This can be used to convert the Condition flags to a Boolean value or mask in a general-purpose register, for example. These instructions include `CSET` and `CSETM`.
- Conditional compare. This instruction type sets the Condition flags to the result of a comparison if the original condition is true, otherwise it sets the Condition flags to an immediate value. It permits the flattening of nested conditional expressions without using conditional branches or performing Boolean arithmetic within the general-purpose registers. The conditional compare instructions are `CCMP` and `CCMN`.

The A64 base instructions that update the Condition flags as an output are:

- Flag-setting data processing instructions, such as `ADCS`, `ADDS`, `ANDS`, `BICS`, `RMIF`, `SBCS`, `SETF8`, `SETF16`, and `SUBS`, and the aliases `CMN`, `CMP`, and `TST`.
- Conditional compare instructions such as `CCMN`, `CCMP`.
- The random number generation instructions `MRS RNDR` and `MRS RNDRRS`, see *Effect of random number generation instructions on Condition flags*.

The A64 base instructions that manipulate the Condition flags are:

- The flag manipulation instruction `CFINV`, which inverts the value of the Carry flag.

- If FEAT_FlagM2 is implemented, the base instructions AXFLAG and XAFLAG. These instructions convert between the Arm floating point comparison PSTATE condition flag format and an alternative format shown in Table C6-1.

Table C6-1 Relationship between ARM format and alternative format PSTATE condition flags

	ARM format				Alternative format			
Result	N	Z	C	V	N	Z	C	V
Greater than	0	0	1	0	0	0	1	0
Less than	1	0	0	0	0	0	0	0
Equal	0	1	1	0	0	1	1	0
Unordered	0	0	1	1	0	1	0	0

The flags can be directly accessed for a read/write using the *NZCV, Condition Flags*.

The A64 base instructions also include conditional branch instructions that do not use the Condition flags as an input:

- Compare and branch if a register is zero or nonzero, CBZ and CBNZ.
- Test a single bit in a register and branch if the bit is zero or nonzero, TBZ and TBNZ.

Effect of random number generation instructions on Condition flags

If FEAT_RNG is implemented, then:

- When a valid random number is returned, the PSTATE.NZCV flags are set to 0b0000.
- If the random number hardware is not capable of returning a random number in a reasonable period of time, the PSTATE.NZCV flags are set to 0b0100, and the random number generation instructions return the value 0.

Note

The definition of "reasonable period of time" is IMPLEMENTATION DEFINED. The expectation is that software might use this as an opportunity to reschedule or run a different routine, perhaps after a small number of retries have failed to return a valid value.

C6.2 Alphabetical list of A64 base instructions

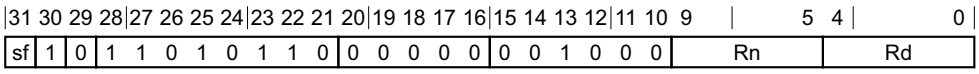
This section lists every instruction in the base category of the A64 instruction set. For details of the format used, see *Understanding the A64 instruction descriptions*.

C6.2.1 ABS

Absolute value computes the absolute value of the signed integer value in the source register, and writes the result to the destination register.

Integer

(FEAT_CSSC)



32-bit variant

Applies when `sf == 0`.

```
ABS <Wd>, <Wn>
```

64-bit variant

Applies when `sf == 1`.

```
ABS <Xd>, <Xn>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_CSSC) then UNDEFINED;
constant integer datasize = 32 << UInt(sf);
integer n = UInt(Rn);
integer d = UInt(Rd);
```

Assembler symbols

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

Operation

```
bits(datasize) operand1 = X[n, datasize];
integer result = Abs(SInt(operand1));
X[d, datasize] = result<datasize-1:0>;
```

Operational information

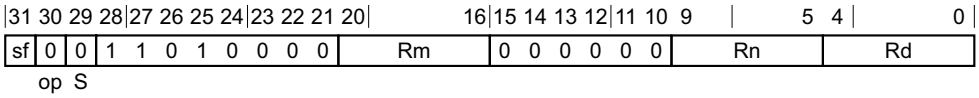
If PSTATE.DIT is 1:

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

— The values of the NZCV flags.

C6.2.2 ADC

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



32-bit variant

Applies when `sf == 0`.

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

64-bit variant

Applies when `sf == 1`.

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

Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
constant integer datasize = 32 << UInt(sf);
```

Assembler symbols

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

Operation

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

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

X[d, datasize] = result;
```

Operational information

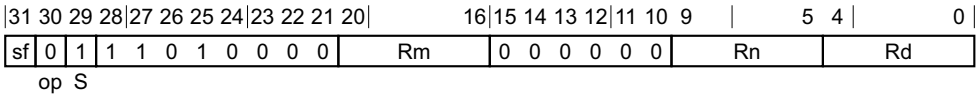
If PSTATE.DIT is 1:

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

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

C6.2.3 ADCS

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



32-bit variant

Applies when `sf == 0`.

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

64-bit variant

Applies when `sf == 1`.

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

Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
constant integer datasize = 32 << UInt(sf);
```

Assembler symbols

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

Operation

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

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

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

X[d, datasize] = result;
```

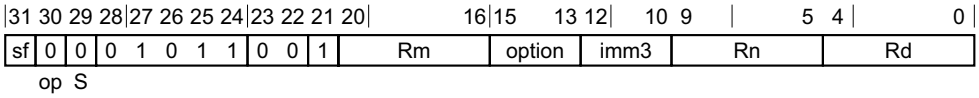
Operational information

If PSTATE.DIT is 1:

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

C6.2.4 ADD (extended register)

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



32-bit variant

Applies when sf == 0.

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

64-bit variant

Applies when sf == 1.

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

Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
constant integer datasize = 32 << UInt(sf);
ExtendType extend_type = DecodeRegExtend(option);
integer shift = UInt(imm3);
if shift > 4 then UNDEFINED;
```

Assembler symbols

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

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

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

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

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

<R> Is a width specifier, encoded in the "option" field. It can have the following values:

W	when option = 00x
W	when option = 010
X	when option = x11
W	when option = 10x
W	when option = 110

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

<extend> For the 32-bit variant: is the extension to be applied to the second source operand, encoded in the "option" field. It can have the following values:

UXTB when option = 000

UXTH when option = 001

LSL|UXTW when option = 010

UXTX when option = 011

SXTB when option = 100

SXTH when option = 101

SXTW when option = 110

SXTX when option = 111

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

For the 64-bit variant: is the extension to be applied to the second source operand, encoded in the "option" field. It can have the following values:

UXTB when option = 000

UXTH when option = 001

UXTW when option = 010

LSL|UXTX when option = 011

SXTB when option = 100

SXTH when option = 101

SXTW when option = 110

SXTX when option = 111

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

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

Operation

```
bits(datasize) result;
bits(datasize) operand1 = if n == 31 then SP[]<datasize-1:0> else X[n, datasize];
bits(datasize) operand2 = ExtendReg(m, extend_type, shift, datasize);

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

if d == 31 then
    SP[] = ZeroExtend(result, 64);
else
    X[d, datasize] = result;
```

Operational information

If PSTATE.DIT is 1:

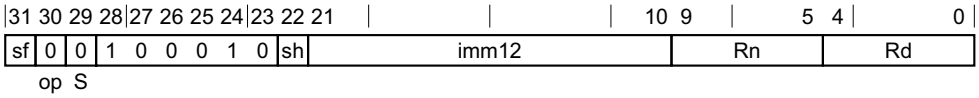
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

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

C6.2.5 ADD (immediate)

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

This instruction is used by the alias MOV (to/from SP). See *Alias conditions* for details of when each alias is preferred.



32-bit variant

Applies when `sf == 0`.

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

64-bit variant

Applies when `sf == 1`.

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

Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
constant integer datasize = 32 << UInt(sf);
bits(datasize) imm;

case sh of
    when '0' imm = ZeroExtend(imm12, datasize);
    when '1' imm = ZeroExtend(imm12:Zeros(12), datasize);
```

Alias conditions

Alias	is preferred when
MOV (to/from SP)	`sh == '0' && imm12 == '000000000000' && (Rd == '11111' || Rn == '11111')`

Assembler symbols

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

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

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

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

`<imm>` Is an unsigned immediate, in the range 0 to 4095, encoded in the "imm12" field.

<shift> Is the optional left shift to apply to the immediate, defaulting to LSL #0 and encoded in the "sh" field. It can have the following values:

LSL #0 when sh = 0

LSL #12 when sh = 1

Operation

```
bits(datasize) result;
bits(datasize) operand1 = if n == 31 then SP[]<datasize-1:0> else X[n, datasize];

(result, -) = AddWithCarry(operand1, imm, '0');

if d == 31 then
    SP[] = ZeroExtend(result, 64);
else
    X[d, datasize] = result;
```

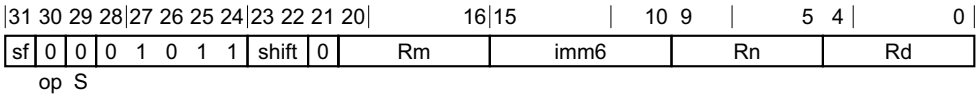
Operational information

If PSTATE.DIT is 1:

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

C6.2.6 ADD (shifted register)

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



32-bit variant

Applies when `sf == 0`.

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

64-bit variant

Applies when `sf == 1`.

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

Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
constant integer datasize = 32 << UInt(sf);

if shift == '11' then UNDEFINED;
if sf == '0' && imm6<5> == '1' then UNDEFINED;

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

Assembler symbols

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

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

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

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

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

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

`<shift>` Is the optional shift type to be applied to the second source operand, defaulting to LSL and encoded in the "shift" field. It can have the following values:

- `LSL` when `shift = 00`
- `LSR` when `shift = 01`
- `ASR` when `shift = 10`

The encoding `shift = 11` is reserved.

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

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

Operation

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

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

X[d, datasize] = result;
```

Operational information

If PSTATE.DIT is 1:

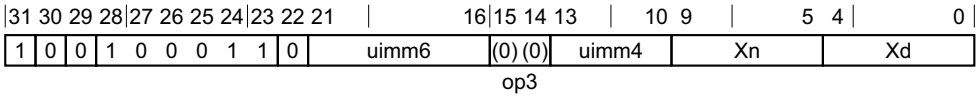
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.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

C6.2.7 ADDG

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

Integer

(FEAT_MTE)



Encoding

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

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_MTE) then UNDEFINED;
integer d = UInt(Xd);
integer n = UInt(Xn);
bits(64) offset = LSL(ZeroExtend(uimm6, 64), LOG2_TAG_GRANULE);
```

Assembler symbols

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

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

<uimm6> Is an unsigned immediate, a multiple of 16 in the range 0 to 1008, encoded in the "uimm6" field.

<uimm4> Is an unsigned immediate, in the range 0 to 15, encoded in the "uimm4" field.

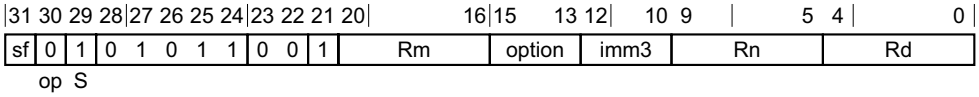
Operation

```
bits(64) operand1 = if n == 31 then SP[] else X[n, 64];
bits(4) start_tag = AArch64.AllocationTagFromAddress(operand1);
bits(16) exclude = GCR_EL1.Exclude;
bits(64) result;
bits(4) rtag;

if AArch64.AllocationTagAccessIsEnabled(PSTATE.EL) then
    rtag = AArch64.ChooseNonExcludedTag(start_tag, uimm4, exclude);
else
    rtag = '0000';

(result, -) = AddWithCarry(operand1, offset, '0');
result = AArch64.AddressWithAllocationTag(result, rtag);

if d == 31 then
    SP[] = result;
else
    X[d, 64] = result;
```

C6.2.8 ADDS (extended register)

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

This instruction is used by the alias CMN (extended register). See Alias conditions for details of when each alias is preferred.



32-bit variant

Applies when sf == 0.

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

64-bit variant

Applies when sf == 1.

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

Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
constant integer datasize = 32 << UInt(sf);
ExtendType extend_type = DecodeRegExtend(option);
integer shift = UInt(imm3);
if shift > 4 then UNDEFINED;
```

Alias conditions

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

Assembler symbols

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

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

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

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

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

<R> Is a width specifier, encoded in the "option" field. It can have the following values:

W when option = 00x

W	when option = 010
X	when option = x11
W	when option = 10x
W	when option = 110

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

<extend> For the 32-bit variant: is the extension to be applied to the second source operand, encoded in the "option" field. It can have the following values:

UXTB	when option = 000
UXTH	when option = 001
LSL|UXTW	when option = 010
UXTX	when option = 011
SXTB	when option = 100
SXTH	when option = 101
SXTW	when option = 110
SXTX	when option = 111

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

For the 64-bit variant: is the extension to be applied to the second source operand, encoded in the "option" field. It can have the following values:

UXTB	when option = 000
UXTH	when option = 001
UXTW	when option = 010
LSL|UXTX	when option = 011
SXTB	when option = 100
SXTH	when option = 101
SXTW	when option = 110
SXTX	when option = 111

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

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

Operation

```
bits(datasize) result;
bits(datasize) operand1 = if n == 31 then SP[]<datasize-1:0> else X[n, datasize];
bits(datasize) operand2 = ExtendReg(m, extend_type, shift, datasize);
bits(4) nzcv;

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

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

X[d, datasize] = result;
```

Operational information

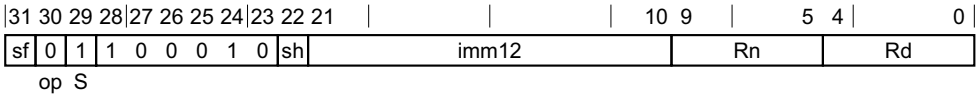
If PSTATE.DIT is 1:

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

C6.2.9 ADDS (immediate)

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

This instruction is used by the alias CMN (immediate). See *Alias conditions* for details of when each alias is preferred.



32-bit variant

Applies when `sf == 0`.

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

64-bit variant

Applies when `sf == 1`.

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

Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
constant integer datasize = 32 << UInt(sf);
bits(datasize) imm;

case sh of
    when '0' imm = ZeroExtend(imm12, datasize);
    when '1' imm = ZeroExtend(imm12:Zeros(12), datasize);
```

Alias conditions

Alias	is preferred when
CMN (immediate)	`Rd == '11111'`

Assembler symbols

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

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

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

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

`<imm>` Is an unsigned immediate, in the range 0 to 4095, encoded in the "imm12" field.

`<shift>` Is the optional left shift to apply to the immediate, defaulting to LSL #0 and encoded in the "sh" field. It can have the following values:

`LSL #0` when `sh = 0`

`LSL #12` when `sh = 1`

Operation

```
bits(datasize) result;
bits(datasize) operand1 = if n == 31 then SP[]<datasize-1:0> else X[n, datasize];
bits(4) nzcv;

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

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

X[d, datasize] = result;
```

Operational information

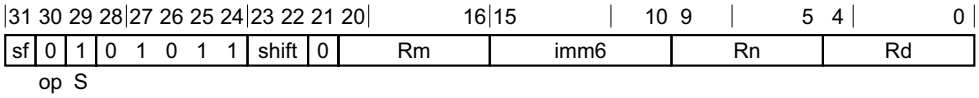
If PSTATE.DIT is 1:

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

C6.2.10 ADDS (shifted register)

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

This instruction is used by the alias CMN (shifted register). See *Alias conditions* for details of when each alias is preferred.



32-bit variant

Applies when `sf == 0`.

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

64-bit variant

Applies when `sf == 1`.

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

Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
constant integer datasize = 32 << UInt(sf);

if shift == '11' then UNDEFINED;
if sf == '0' && imm6<5> == '1' then UNDEFINED;

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

Alias conditions

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

Assembler symbols

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

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

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

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

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

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

<shift> Is the optional shift type to be applied to the second source operand, defaulting to LSL and encoded in the "shift" field. It can have the following values:

LSL when shift = 00

LSR when shift = 01

ASR when shift = 10

The encoding shift = 11 is reserved.

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

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

Operation

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

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

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

X[d, datasize] = result;
```

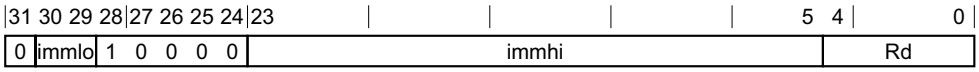
Operational information

If PSTATE.DIT is 1:

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

C6.2.11 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.



Encoding

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

Decode for this encoding

```
integer d = UInt(Rd);
bits(64) imm;

imm = SignExtend(immhi:immlo, 64);
```

Assembler symbols

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

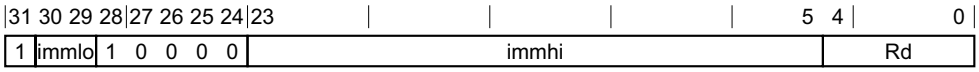
<label> Is the program label whose address is to be calculated. Its offset from the address of this instruction, in the range +/-1MB, is encoded in "immhi:immlo".

Operation

```
X[d, 64] = PC64 + imm;
```

C6.2.12 ADRP

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



Encoding

```
ADRP <Xd>, <label>
```

Decode for this encoding

```
integer d = UInt(Rd);
bits(64) imm;

imm = SignExtend(immhi:immlo:Zeros(12), 64);
```

Assembler symbols

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

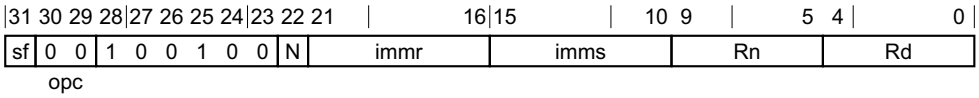
<label> Is the program label whose 4KB page address is to be calculated. Its offset from the page address of this instruction, in the range +/-4GB, is encoded as "immhi:immlo" times 4096.

Operation

```
bits(64) base = PC64<63:12>:Zeros(12);
X[d, 64] = base + imm;
```

C6.2.13 AND (immediate)

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



32-bit variant

Applies when `sf == 0 && N == 0`.

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

64-bit variant

Applies when `sf == 1`.

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

Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
constant integer datasize = 32 << UInt(sf);
bits(datasize) imm;
if sf == '0' && N != '0' then UNDEFINED;
(imm, -) = DecodeBitMasks(N, imms, immr, TRUE, datasize);
```

Assembler symbols

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

Operation

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

result = operand1 AND imm;
if d == 31 then
    SP[] = ZeroExtend(result, 64);
else
    X[d, datasize] = result;
```

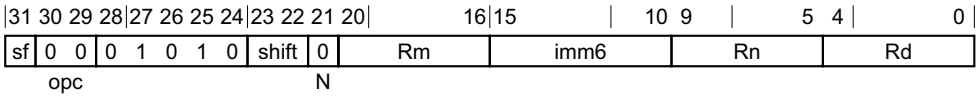
Operational information

If PSTATE.DIT is 1:

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

C6.2.14 AND (shifted register)

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



32-bit variant

Applies when `sf == 0`.

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

64-bit variant

Applies when `sf == 1`.

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

Decode for all variants of this encoding

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

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

Assembler symbols

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

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

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

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

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

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

`<shift>` Is the optional shift to be applied to the final source, defaulting to LSL and encoded in the "shift" field. It can have the following values:

- `LSL` when `shift = 00`
- `LSR` when `shift = 01`
- `ASR` when `shift = 10`
- `ROR` when `shift = 11`

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

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

Operation

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

result = operand1 AND operand2;
X[d, datasize] = result;
```

Operational information

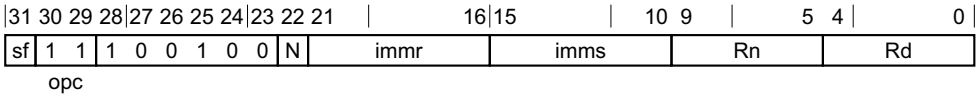
If PSTATE.DIT is 1:

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

C6.2.15 ANDS (immediate)

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

This instruction is used by the alias TST (immediate). See *Alias conditions* for details of when each alias is preferred.



32-bit variant

Applies when `sf == 0 && N == 0`.

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

64-bit variant

Applies when `sf == 1`.

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

Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
constant integer datasize = 32 << UInt(sf);

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

Alias conditions

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

Assembler symbols

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

Operation

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

result = operand1 AND imm;
PSTATE.<N,Z,C,V> = result<datasize-1>:IsZeroBit(result):'00';

X[d, datasize] = result;
```

Operational information

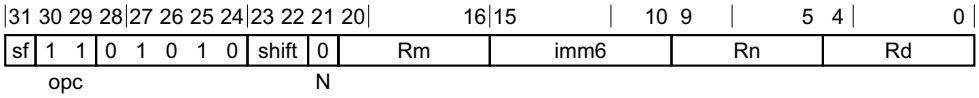
If PSTATE.DIT is 1:

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

C6.2.16 ANDS (shifted register)

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

This instruction is used by the alias TST (shifted register). See *Alias conditions* for details of when each alias is preferred.



32-bit variant

Applies when `sf == 0`.

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

64-bit variant

Applies when `sf == 1`.

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

Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
constant integer datasize = 32 << UInt(sf);

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

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

Alias conditions

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

Assembler symbols

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

<shift> Is the optional shift to be applied to the final source, defaulting to LSL and encoded in the "shift" field. It can have the following values:

LSL when shift = 00

LSR when shift = 01

ASR when shift = 10

ROR when shift = 11

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

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

Operation

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

result = operand1 AND operand2;
PSTATE.<N,Z,C,V> = result<datasize-1>:IsZeroBit(result):'00';

X[d, datasize] = result;
```

Operational information

If PSTATE.DIT is 1:

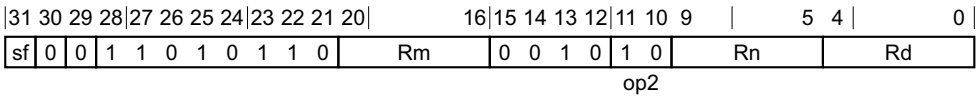
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.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

C6.2.17 ASR (register)

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

This instruction is an alias of the ASRV instruction. This means that:

- The encodings in this description are named to match the encodings of ASRV.
- The description of ASRV gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.



32-bit variant

Applies when `sf == 0`.

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

is equivalent to

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

and is always the preferred disassembly.

64-bit variant

Applies when `sf == 1`.

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

is equivalent to

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

and is always the preferred disassembly.

Assembler symbols

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

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

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

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

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

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

Operation

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

Operational information

If PSTATE.DIT is 1:

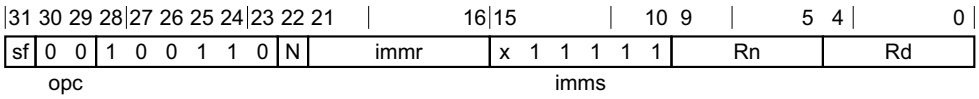
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.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

C6.2.18 ASR (immediate)

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

This instruction is an alias of the SBFM instruction. This means that:

- The encodings in this description are named to match the encodings of SBFM.
- The description of SBFM gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.



32-bit variant

Applies when `sf == 0 && N == 0 && imms == 011111`.

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

is equivalent to

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

and is always the preferred disassembly.

64-bit variant

Applies when `sf == 1 && N == 1 && imms == 111111`.

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

is equivalent to

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

and is always the preferred disassembly.

Assembler symbols

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

Operation

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

Operational information

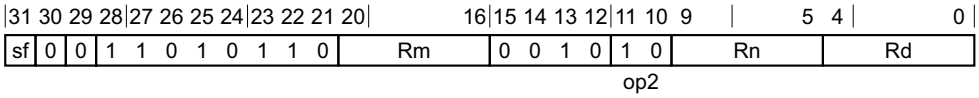
If PSTATE.DIT is 1:

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

C6.2.19 ASRV

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

This instruction is used by the alias ASR (register). The alias is always the preferred disassembly.



32-bit variant

Applies when `sf == 0`.

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

64-bit variant

Applies when `sf == 1`.

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

Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
constant integer datasize = 32 << UInt(sf);
ShiftType shift_type = DecodeShift(op2);
```

Assembler symbols

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

Operation

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

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

Operational information

If PSTATE.DIT is 1:

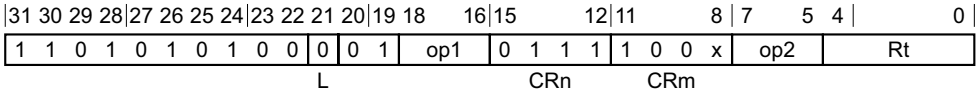
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.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

C6.2.20 AT

Address Translate. For more information, see *op0==0b01, cache maintenance, TLB maintenance, address translation, prediction restriction, BRBE, Trace Extension, and Guarded Control Stack instructions*.

This instruction is an alias of the SYS instruction. This means that:

- The encodings in this description are named to match the encodings of SYS.
- The description of SYS gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.



Encoding

`AT <at_op>, <Xt>`

is equivalent to

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

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

Assembler symbols

`<at_op>` Is an AT instruction name, as listed for the AT system instruction group, encoded in the "op1:CRm<0>:op2" field. It can have the following values:

`S1E1R` when `op1 = 000`, `CRm<0> = 0`, `op2 = 000`

`S1E1W` when `op1 = 000`, `CRm<0> = 0`, `op2 = 001`

`S1E0R` when `op1 = 000`, `CRm<0> = 0`, `op2 = 010`

`S1E0W` when `op1 = 000`, `CRm<0> = 0`, `op2 = 011`

`S1E2R` when `op1 = 100`, `CRm<0> = 0`, `op2 = 000`

`S1E2W` when `op1 = 100`, `CRm<0> = 0`, `op2 = 001`

`S12E1R` when `op1 = 100`, `CRm<0> = 0`, `op2 = 100`

`S12E1W` when `op1 = 100`, `CRm<0> = 0`, `op2 = 101`

`S12E0R` when `op1 = 100`, `CRm<0> = 0`, `op2 = 110`

`S12E0W` when `op1 = 100`, `CRm<0> = 0`, `op2 = 111`

`S1E3R` when `op1 = 110`, `CRm<0> = 0`, `op2 = 000`

`S1E3W` when `op1 = 110`, `CRm<0> = 0`, `op2 = 001`

When FEAT_PAN2 is implemented, the following values are also valid:

`S1E1RP` when `op1 = 000`, `CRm<0> = 1`, `op2 = 000`

`S1E1WP` when `op1 = 000`, `CRm<0> = 1`, `op2 = 001`

When FEAT_ATS1A is implemented, the following values are also valid:

`S1E1A` when `op1 = 000`, `CRm<0> = 1`, `op2 = 010`

`S1E2A` when `op1 = 100`, `CRm<0> = 1`, `op2 = 010`

`S1E3A` when `op1 = 110`, `CRm<0> = 1`, `op2 = 010`

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

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

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

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

Operation

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

C6.2.21 AUTDA, AUTDZA

Authenticate Data address, using key A. This instruction authenticates a data address, using a modifier and key A.

The address is in the general-purpose register that is specified by <Xd>.

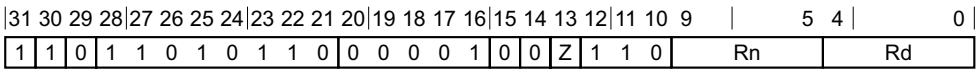
The modifier is:

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

If the authentication passes, the upper bits of the address are restored to enable subsequent use of the address. For information on behavior if the authentication fails, see *Faulting on pointer authentication*.

Integer

(FEAT_PAuth)



AUTDA variant

Applies when Z == 0.

```
AUTDA <Xd>, <Xn|SP>
```

AUTDZA variant

Applies when Z == 1 && Rn == 11111.

```
AUTDZA <Xd>
```

Decode for all variants of this encoding

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

if !IsFeatureImplemented(FEAT_PAuth) then
    UNDEFINED;

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

Assembler symbols

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

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

Operation

```
if IsFeatureImplemented(FEAT_PAuth) then
    if source_is_sp then
        X[d, 64] = AuthDA(X[d, 64], SP[], FALSE);
    else
        X[d, 64] = AuthDA(X[d, 64], X[n, 64], FALSE);
```

C6.2.22 AUTDB, AUTDZB

Authenticate Data address, using key B. This instruction authenticates a data address, using a modifier and key B.

The address is in the general-purpose register that is specified by <Xd>.

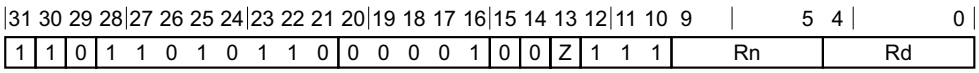
The modifier is:

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

If the authentication passes, the upper bits of the address are restored to enable subsequent use of the address. For information on behavior if the authentication fails, see *Faulting on pointer authentication*.

Integer

(FEAT_PAuth)



AUTDB variant

Applies when Z == 0.

```
AUTDB <Xd>, <Xn|SP>
```

AUTDZB variant

Applies when Z == 1 && Rn == 11111.

```
AUTDZB <Xd>
```

Decode for all variants of this encoding

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

if !IsFeatureImplemented(FEAT_PAuth) then
    UNDEFINED;

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

Assembler symbols

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

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

Operation

```
if IsFeatureImplemented(FEAT_PAuth) then
    if source_is_sp then
        X[d, 64] = AuthDB(X[d, 64], SP[], FALSE);
    else
        X[d, 64] = AuthDB(X[d, 64], X[n, 64], FALSE);
```

C6.2.23 AUTIA, AUTIA1716, AUTIASP, AUTIAZ, AUTIZA

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

If the authentication passes, the upper bits of the address are restored to enable subsequent use of the address. For information on behavior if the authentication fails, see *Faulting on pointer authentication*.

The address is:

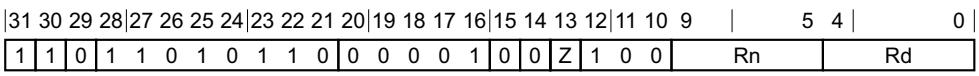
- In the general-purpose register that is specified by <Xd> for AUTIA and AUTIZA.
- In X17, for AUTIA1716.
- In X30, for AUTIASP and AUTIAZ.

The modifier is:

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

Integer

(FEAT_PAuth)



AUTIA variant

Applies when Z == 0.

```
AUTIA <Xd>, <Xn|SP>
```

AUTIZA variant

Applies when Z == 1 && Rn == 11111.

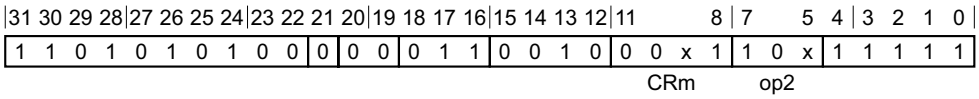
```
AUTIZA <Xd>
```

Decode for all variants of this encoding

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

if !IsFeatureImplemented(FEAT_PAuth) then
    UNDEFINED;

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

System

(FEAT_PAuth)



AUTIA1716 variant

Applies when `CRm == 0001 && op2 == 100`.

```
AUTIA1716
```

AUTIASP variant

Applies when `CRm == 0011 && op2 == 101`.

```
AUTIASP
```

AUTIAZ variant

Applies when `CRm == 0011 && op2 == 100`.

```
AUTIAZ
```

Decode for all variants of this encoding

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

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

Assembler symbols

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

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

Operation for all encodings

```
if IsFeatureImplemented(FEAT_PAuth) then
    if source_is_sp then
        X[d, 64] = AuthIA(X[d, 64], SP[], FALSE);
    else
        X[d, 64] = AuthIA(X[d, 64], X[n, 64], FALSE);
```

C6.2.24 AUTIB, AUTIB1716, AUTIBSP, AUTIBZ, AUTIZB

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

If the authentication passes, the upper bits of the address are restored to enable subsequent use of the address. For information on behavior if the authentication fails, see *Faulting on pointer authentication*.

The address is:

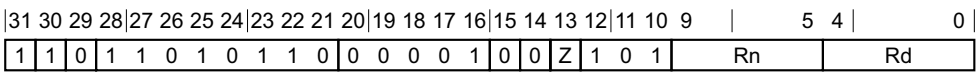
- In the general-purpose register that is specified by <Xd> for AUTIB and AUTIZB.
- In X17, for AUTIB1716.
- In X30, for AUTIBSP and AUTIBZ.

The modifier is:

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

Integer

(FEAT_PAuth)



AUTIB variant

Applies when Z == 0.

```
AUTIB <Xd>, <Xn|SP>
```

AUTIZB variant

Applies when Z == 1 && Rn == 11111.

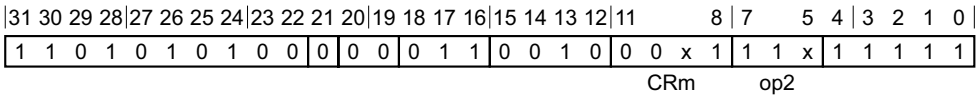
```
AUTIZB <Xd>
```

Decode for all variants of this encoding

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

if !IsFeatureImplemented(FEAT_PAuth) then
    UNDEFINED;

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

System

(FEAT_PAuth)



AUTIB1716 variant

Applies when `CRm == 0001 && op2 == 110`.

```
AUTIB1716
```

AUTIBSP variant

Applies when `CRm == 0011 && op2 == 111`.

```
AUTIBSP
```

AUTIBZ variant

Applies when `CRm == 0011 && op2 == 110`.

```
AUTIBZ
```

Decode for all variants of this encoding

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

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

Assembler symbols

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

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

Operation for all encodings

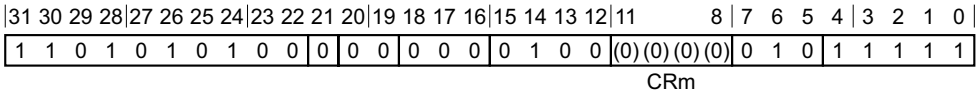
```
if IsFeatureImplemented(FEAT_PAuth) then
    if source_is_sp then
        X[d, 64] = AuthIB(X[d, 64], SP[], FALSE);
    else
        X[d, 64] = AuthIB(X[d, 64], X[n, 64], FALSE);
```

C6.2.25 AXFLAG

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

System

(FEAT_FlagM2)



Encoding

AXFLAG

Decode for this encoding

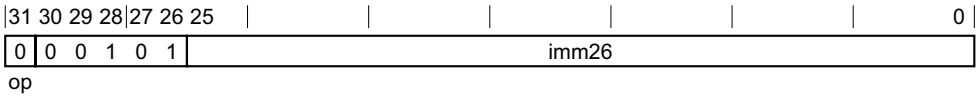
```
if !IsFeatureImplemented(FEAT_FlagM2) then UNDEFINED;
```

Operation

```
bit z = PSTATE.Z OR PSTATE.V;
bit c = PSTATE.C AND NOT(PSTATE.V);

PSTATE.N = '0';
PSTATE.Z = z;
PSTATE.C = c;
PSTATE.V = '0';
```

C6.2.26 B

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



Encoding

```
B <label>
```

Decode for this encoding

```
bits(64) offset = SignExtend(imm26:'00', 64);
```

Assembler symbols

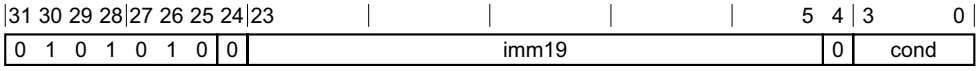
<label> Is the program label to be unconditionally branched to. Its offset from the address of this instruction, in the range +/-128MB, is encoded as "imm26" times 4.

Operation

```
BranchTo(PC64 + offset, BranchType_DIR, FALSE);
```

C6.2.27 B.cond

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



Encoding

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

Decode for this encoding

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

Assembler symbols

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

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

Operation

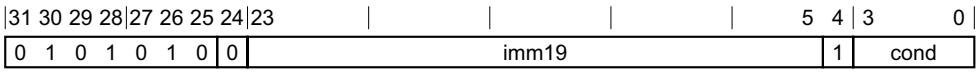
```
if ConditionHolds(cond) then
    BranchTo(PC64 + offset, BranchType_DIR, TRUE);
else
    BranchNotTaken(BranchType_DIR, TRUE);
```

C6.2.28 BC.cond

Branch Consistent conditionally to a label at a PC-relative offset, with a hint that this branch will behave very consistently and is very unlikely to change direction.

19-bit signed PC-relative branch offset

(FEAT_HBC)



Encoding

```
BC.<cond> <label>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_HBC) then UNDEFINED;
bits(64) offset = SignExtend(imm19:'00', 64);
```

Assembler symbols

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

Operation

```
if ConditionHolds(cond) then
    BranchTo(PC64 + offset, BranchType_DIR, TRUE);
else
    BranchNotTaken(BranchType_DIR, TRUE);
```

C6.2.29 BFC

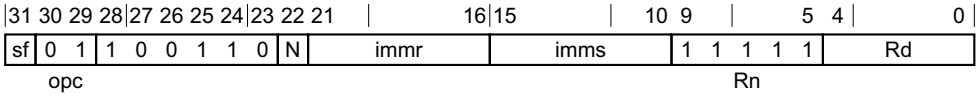
Bitfield Clear sets a bitfield of <width> bits at bit position <lsb> of the destination register to zero, leaving the other destination bits unchanged.

This instruction is an alias of the BFM instruction. This means that:

- The encodings in this description are named to match the encodings of BFM.
- The description of BFM gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Leaving other bits unchanged

(FEAT_ASMv8p2)



32-bit variant

Applies when `sf == 0 && N == 0`.

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

is equivalent to

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

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

64-bit variant

Applies when `sf == 1 && N == 1`.

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

is equivalent to

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

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

Assembler symbols

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

Operation

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

Operational information

If PSTATE.DIT is 1:

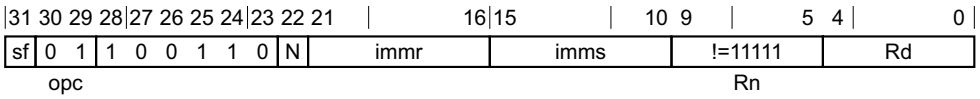
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.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

C6.2.30 BFI

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

This instruction is an alias of the BFM instruction. This means that:

- The encodings in this description are named to match the encodings of BFM.
- The description of BFM gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.



32-bit variant

Applies when sf == 0 && N == 0.

```
BFI <Wd>, <Wn>, #<lsb>, #<width>
```

is equivalent to

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

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

64-bit variant

Applies when sf == 1 && N == 1.

```
BFI <Xd>, <Xn>, #<lsb>, #<width>
```

is equivalent to

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

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

Assembler symbols

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

Operation

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

Operational information

If PSTATE.DIT is 1:

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

C6.2.31 BFM

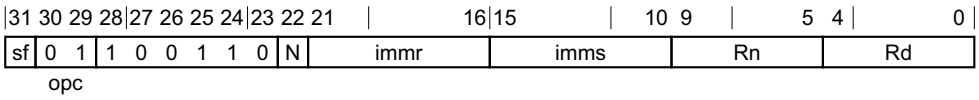
Bitfield Move is usually accessed via one of its aliases, which are always preferred for disassembly.

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

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

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

This instruction is used by the aliases BFC, BFI, and BFXIL. See *Alias conditions* for details of when each alias is preferred.



32-bit variant

Applies when sf == 0 && N == 0.

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

64-bit variant

Applies when sf == 1 && N == 1.

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

Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
constant integer datasize = 32 << UInt(sf);

integer r;
bits(datasize) wmask;
bits(datasize) tmask;

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

r = UInt(immr);
(wmask, tmask) = DecodeBitMasks(N, imms, immr, FALSE, datasize);
```

Alias conditions

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

Assembler symbols

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

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

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

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

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

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

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

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

Operation

```
bits(datasize) dst = X[d, datasize];
bits(datasize) src = X[n, datasize];

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

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

Operational information

If PSTATE.DIT is 1:

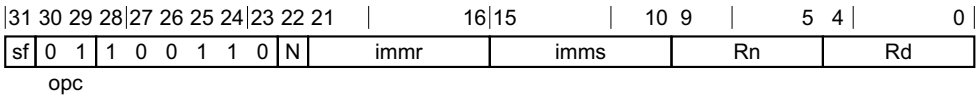
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.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

C6.2.32 BFXIL

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

This instruction is an alias of the BFM instruction. This means that:

- The encodings in this description are named to match the encodings of BFM.
- The description of BFM gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.



32-bit variant

Applies when sf == 0 && N == 0.

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

is equivalent to

```
BFM  <Wd>, <Wn>, #<lsb>, #(<lsb>+<width>-1)
```

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

64-bit variant

Applies when sf == 1 && N == 1.

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

is equivalent to

```
BFM  <Xd>, <Xn>, #<lsb>, #(<lsb>+<width>-1)
```

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

Assembler symbols

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

Operation

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

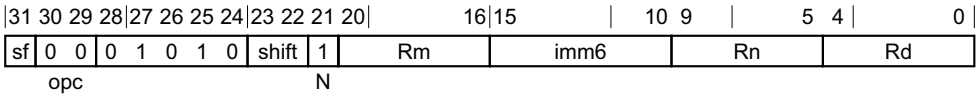
Operational information

If PSTATE.DIT is 1:

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

C6.2.33 BIC (shifted register)

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



32-bit variant

Applies when `sf == 0`.

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

64-bit variant

Applies when `sf == 1`.

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

Decode for all variants of this encoding

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

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

Assembler symbols

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

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

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

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

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

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

`<shift>` Is the optional shift to be applied to the final source, defaulting to LSL and encoded in the "shift" field. It can have the following values:

- `LSL` when `shift = 00`
- `LSR` when `shift = 01`
- `ASR` when `shift = 10`
- `ROR` when `shift = 11`

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

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

Operation

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

operand2 = NOT(operand2);

result = operand1 AND operand2;
X[d, datasize] = result;
```

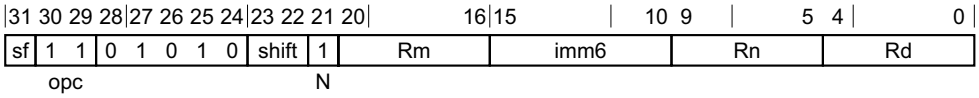
Operational information

If PSTATE.DIT is 1:

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

C6.2.34 BICS (shifted register)

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



32-bit variant

Applies when `sf == 0`.

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

64-bit variant

Applies when `sf == 1`.

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

Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
constant integer datasize = 32 << UInt(sf);

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

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

Assembler symbols

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

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

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

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

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

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

`<shift>` Is the optional shift to be applied to the final source, defaulting to LSL and encoded in the "shift" field. It can have the following values:

- `LSL` when `shift = 00`
- `LSR` when `shift = 01`
- `ASR` when `shift = 10`
- `ROR` when `shift = 11`

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

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

Operation

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

operand2 = NOT(operand2);

result = operand1 AND operand2;
PSTATE.<N,Z,C,V> = result<datasize-1>:IsZeroBit(result):'00';

X[d, datasize] = result;
```

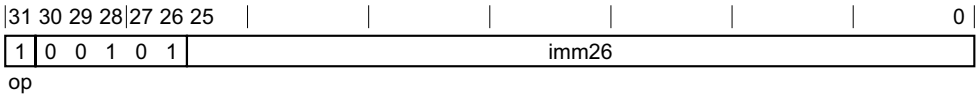
Operational information

If PSTATE.DIT is 1:

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

C6.2.35 BL

Branch with Link branches to a PC-relative offset, setting the register X30 to PC+4. It provides a hint that this is a subroutine call.



Encoding

```
BL <label>
```

Decode for this encoding

```
bits(64) offset = SignExtend(imm26:'00', 64);
```

Assembler symbols

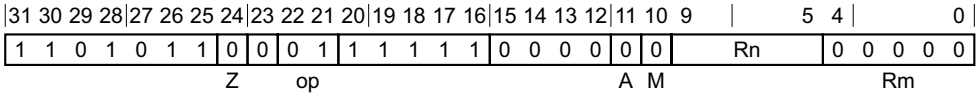
<label> Is the program label to be unconditionally branched to. Its offset from the address of this instruction, in the range +/-128MB, is encoded as "imm26" times 4.

Operation

```
if IsFeatureImplemented(FEAT_GCS) && GCSPCREnabled(PSTATE.EL) then
    AddGCSRecord(PC64 + 4);
X[30, 64] = PC64 + 4;

BranchTo(PC64 + offset, BranchType_DIRCALL, FALSE);
```

C6.2.36 BLR

Branch with Link to Register calls a subroutine at an address in a register, setting register X30 to PC+4.



Encoding

```
BLR <Xn>
```

Decode for this encoding

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

Assembler symbols

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

Operation

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

if IsFeatureImplemented(FEAT_GCS) && GCSPCREnabled(PSTATE.EL) then
    AddGCSRecord(PC64 + 4);
X[30, 64] = PC64 + 4;

// Value in BTypeNext will be used to set PSTATE.BTYPE
BTypeNext = '10';
BranchTo(target, BranchType_INDCALL, FALSE);
```

C6.2.37 BLRAA, BLRAAZ, BLRAB, BLRABZ

Branch with Link to Register, with pointer authentication. This instruction authenticates the address in the general-purpose register that is specified by <Xn>, using a modifier and the specified key, and calls a subroutine at the authenticated address, setting register X30 to PC+4.

The modifier is:

- In the general-purpose register or stack pointer that is specified by <Xm|SP>, for BLRAA and BLRAB.
- The value zero, for BLRAAZ and BLRABZ.

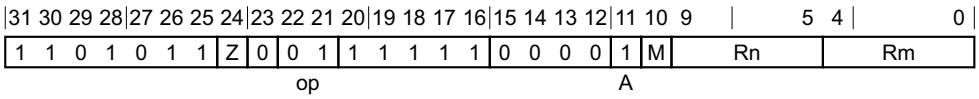
Key A is used for BLRAA and BLRAAZ. Key B is used for BLRAB and BLRABZ.

If the authentication passes, the PE continues execution at the target of the branch. For information on behavior if the authentication fails, see *Faulting on pointer authentication*.

The authenticated address is not written back to the general-purpose register.

Integer

(FEAT_PAuth)



Key A, zero modifier variant

Applies when Z == 0 && M == 0 && Rm == 11111.

```
BLRAAZ <Xn>
```

Key A, register modifier variant

Applies when Z == 1 && M == 0.

```
BLRAA <Xn>, <Xm|SP>
```

Key B, zero modifier variant

Applies when Z == 0 && M == 1 && Rm == 11111.

```
BLRABZ <Xn>
```

Key B, register modifier variant

Applies when Z == 1 && M == 1.

```
BLRAB <Xn>, <Xm|SP>
```

Decode for all variants of this encoding

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

if !IsFeatureImplemented(FEAT_PAuth) then
    UNDEFINED;

if Z == '0' && m != 31 then
    UNDEFINED;
```

Assembler symbols

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

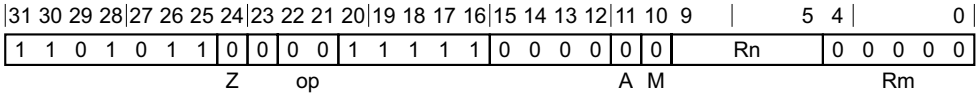
<Xm|SP> Is the 64-bit name of the general-purpose source register or stack pointer holding the modifier, encoded in the "Rm" field.

Operation

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

bits(64) modifier = if source_is_sp then SP[] else X[m, 64];

if use_key_a then
    target = AuthIA(target, modifier, TRUE);
else
    target = AuthIB(target, modifier, TRUE);

if IsFeatureImplemented(FEAT_GCS) && GCSPCREnabled(PSTATE.EL) then
    AddGCSRecord(PC64 + 4);
X[30, 64] = PC64 + 4;

// Value in BTypeNext will be used to set PSTATE.BTYPE
BTypeNext = '10';
BranchTo(target, BranchType_INDCALL, FALSE);
```

C6.2.38 BR

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



Encoding

BR <Xn>

Decode for this encoding

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

Assembler symbols

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

Operation

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

// Value in BTypeNext will be used to set PSTATE.BTYPE
if InGuardedPage then
    if n == 16 || n == 17 then
        BTypeNext = '01';
    else
        BTypeNext = '11';
else
    BTypeNext = '01';
BranchTo(target, BranchType_INDIR, FALSE);
```

C6.2.39 BRAA, BRAAZ, BRAB, BRABZ

Branch to Register, with pointer authentication. This instruction authenticates the address in the general-purpose register that is specified by <Xn>, using a modifier and the specified key, and branches to the authenticated address.

The modifier is:

- In the general-purpose register or stack pointer that is specified by <Xm|SP>, for BRAA and BRAB.
- The value zero, for BRAAZ and BRABZ.

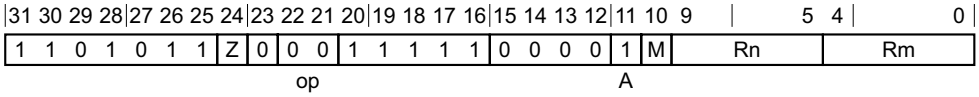
Key A is used for BRAA and BRAAZ. Key B is used for BRAB and BRABZ.

If the authentication passes, the PE continues execution at the target of the branch. For information on behavior if the authentication fails, see *Faulting on pointer authentication*.

The authenticated address is not written back to the general-purpose register.

Integer

(FEAT_PAuth)



Key A, zero modifier variant

Applies when Z == 0 && M == 0 && Rm == 11111.

```
BRAAZ <Xn>
```

Key A, register modifier variant

Applies when Z == 1 && M == 0.

```
BRAA <Xn>, <Xm|SP>
```

Key B, zero modifier variant

Applies when Z == 0 && M == 1 && Rm == 11111.

```
BRABZ <Xn>
```

Key B, register modifier variant

Applies when Z == 1 && M == 1.

```
BRAB <Xn>, <Xm|SP>
```

Decode for all variants of this encoding

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

if !IsFeatureImplemented(FEAT_PAuth) then
    UNDEFINED;

if Z == '0' && m != 31 then
    UNDEFINED;
```

Assembler symbols

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

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

Operation

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

bits(64) modifier = if source_is_sp then SP[] else X[m, 64];

if use_key_a then
    target = AuthIA(target, modifier, TRUE);
else
    target = AuthIB(target, modifier, TRUE);

// Value in BTypeNext will be used to set PSTATE.BTYPE
if InGuardedPage then
    if n == 16 || n == 17 then
        BTypeNext = '01';
    else
        BTypeNext = '11';
else
    BTypeNext = '01';
BranchTo(target, BranchType_INDIR, FALSE);
```

C6.2.40 BRB

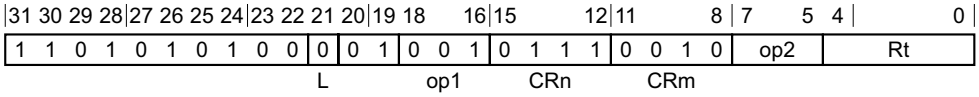
Branch Record Buffer. For more information, see *op0==0b01, cache maintenance, TLB maintenance, address translation, prediction restriction, BRBE, Trace Extension, and Guarded Control Stack instructions*.

This instruction is an alias of the SYS instruction. This means that:

- The encodings in this description are named to match the encodings of SYS.
- The description of SYS gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

System

(FEAT_BRBE)



Encoding

```
BRB <brb_op>{, <Xt>}
```

is equivalent to

```
SYS #1, C7, C2, #<op2>{, <Xt>}
```

and is the preferred disassembly when `SysOp('001','0111','0010',op2) == Sys_BRB`.

Assembler symbols

`<brb_op>` Is a BRB instruction name, as listed for the BRB system instruction group, encoded in the "op2" field. It can have the following values:

`IALL`	when `op2 = 100`
`INJ`	when `op2 = 101`

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

`<Xt>` Is the 64-bit name of the optional general-purpose source register, defaulting to '11111', encoded in the "Rt" field.

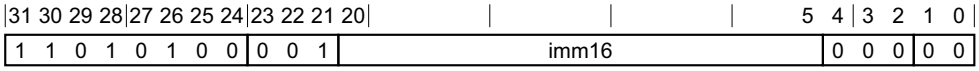
Operation

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

C6.2.41 BRK

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

Within a guarded memory region, while PSTATE.BTYPE != `0b00`, a `BRK` instruction will not generate a Branch Target Exception and will generate a Breakpoint Instruction exception as normal. For more information, see *PSTATE.BTYPE*.



Encoding

```
BRK #<imm>
```

Decode for this encoding

```
if IsFeatureImplemented(FEAT_BTI) then
    SetBTypeCompatible(TRUE);
```

Assembler symbols

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

Operation

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

C6.2.42 BTI

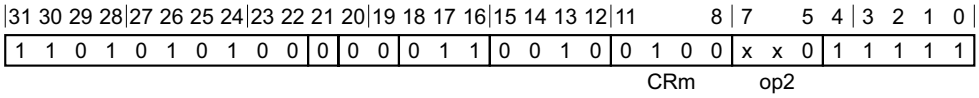
Branch Target Identification. A BTI instruction is used to guard against the execution of instructions that are not the intended target of a branch.

Outside of a guarded memory region, a BTI instruction executes as a NOP. Within a guarded memory region, while PSTATE.BTYPE != 0b00, a BTI instruction compatible with the current value of PSTATE.BTYPE will not generate a Branch Target Exception and will allow execution of subsequent instructions within the memory region. For more information, see *PSTATE.BTYPE*.

The operand <targets> passed to a BTI instruction determines the values of PSTATE.BTYPE that the BTI instruction is compatible with.

System

(FEAT_BTI)



Encoding

```
BTI {<targets>}
```

Decode for this encoding

```
SystemHintOp op;

if CRm:op2 == '0100 xx0' then
    op = SystemHintOp_BTI;
    // Check branch target compatibility between BTI instruction and PSTATE.BTYPE
    SetBTypeCompatible(BTypeCompatible_BTI(op2<2:1>));
else
    EndOfInstruction();
```

Assembler symbols

<targets> Is the type of indirection, encoded in the "op2<2:1>" field. It can have the following values:

(omitted)	when op2<2:1> = 00
c	when op2<2:1> = 01
j	when op2<2:1> = 10
jc	when op2<2:1> = 11

Operation

```
case op of
    when SystemHintOp_YIELD
        Hint_Yield();

    when SystemHintOp_DGH
        Hint_DGH();

    when SystemHintOp_WFE
        integer localtimeout = 1 << 64;    // No local timeout event is generated
        Hint_WFE(localtimeout, WFxType_WFE);

    when SystemHintOp_WFI
        integer localtimeout = 1 << 64;    // No local timeout event is generated
        Hint_WFI(localtimeout, WFxType_WFI);
```

```
when SystemHintOp_SEV
    SendEvent();

when SystemHintOp_SEVL
    SendEventLocal();

when SystemHintOp_ESB
    if IsFeatureImplemented(FEAT_TME) && TSTATE.depth > 0 then
        FailTransaction(TMFailure_ERR, FALSE);
    SynchronizeErrors();
    AArch64.ESBOperation();
    if PSTATE.EL IN {EL0, EL1} && EL2Enabled() then AArch64.vESBOperation();
    TakeUnmaskedSErrorInterrupts();

when SystemHintOp_PSB
    ProfilingSynchronizationBarrier();

when SystemHintOp_TSB
    TraceSynchronizationBarrier();

when SystemHintOp_GCSB
    GCSSynchronizationBarrier();

when SystemHintOp_CHKFEAT
    X[16, 64] = AArch64.ChkFeat(X[16, 64]);

when SystemHintOp_CSDB
    ConsumptionOfSpeculativeDataBarrier();

when SystemHintOp_CLRBHB
    Hint_CLRBHB();

when SystemHintOp_BTI
    SetBTypeNext('00');

when SystemHintOp_NOP
    return;    // do nothing

otherwise
    Unreachable();
```

C6.2.43 CAS, CASA, CASAL, CASL

Compare and Swap word or doubleword in memory reads a 32-bit word or 64-bit doubleword from memory, and compares it against the value held in a first register. If the comparison is equal, the value in a second register is written to memory. If the write is performed, the read and write occur atomically such that no other modification of the memory location can take place between the read and write.

- CASA and CASAL load from memory with acquire semantics.
- CASL and CASAL store to memory with release semantics.
- CAS has neither acquire nor release semantics.

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

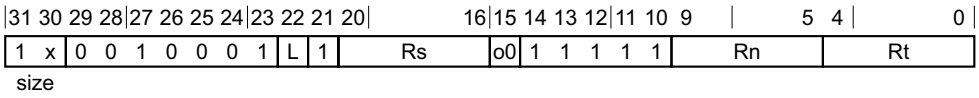
For information about memory accesses, see *Load/store addressing modes*.

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

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

No offset

(FEAT_LSE)



32-bit CAS variant

Applies when size == 10 && L == 0 && o0 == 0.

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

32-bit CASA variant

Applies when size == 10 && L == 1 && o0 == 0.

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

32-bit CASAL variant

Applies when size == 10 && L == 1 && o0 == 1.

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

32-bit CASL variant

Applies when size == 10 && L == 0 && o0 == 1.

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

64-bit CAS variant

Applies when size == 11 && L == 0 && o0 == 0.

```
CAS <Xs>, <Xt>, [<Xn|SP>{,#0}]
```

64-bit CASA variant

Applies when size == 11 && L == 1 && o0 == 0.

```
CASA <Xs>, <Xt>, [<Xn|SP>{,#0}]
```

64-bit CASAL variant

Applies when size == 11 && L == 1 && o0 == 1.

```
CASAL <Xs>, <Xt>, [<Xn|SP>{,#0}]
```

64-bit CASL variant

Applies when size == 11 && L == 0 && o0 == 1.

```
CASL <Xs>, <Xt>, [<Xn|SP>{,#0}]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LSE) then UNDEFINED;

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

constant integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
boolean acquire = L == '1';
boolean release = o0 == '1';
boolean tagchecked = n != 31;
```

Assembler symbols

<Ws>	Is the 32-bit name of the general-purpose register to be compared and loaded, encoded in the "Rs" field.
<Wt>	Is the 32-bit name of the general-purpose register to be conditionally stored, encoded in the "Rt" field.
<Xs>	Is the 64-bit name of the general-purpose register to be compared and loaded, encoded in the "Rs" field.
<Xt>	Is the 64-bit name of the general-purpose register to be conditionally stored, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(datasize) comparevalue;
bits(datasize) newvalue;
bits(datasize) data;

AccessDescriptor accdesc = CreateAccDescAtomicOp(MemAtomicOp_CAS, acquire, release, tagchecked);

comparevalue = X[s, datasize];
newvalue = X[t, datasize];

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

data = MemAtomic(address, comparevalue, newvalue, accdesc);
```

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

C6.2.44 CASB, CASAB, CASALB, CASLB

Compare and Swap byte in memory reads an 8-bit byte from memory, and compares it against the value held in a first register. If the comparison is equal, the value in a second register is written to memory. If the write is performed, the read and write occur atomically such that no other modification of the memory location can take place between the read and write.

- CASAB and CASALB load from memory with acquire semantics.
- CASLB and CASALB store to memory with release semantics.
- CASB has neither acquire nor release semantics.

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

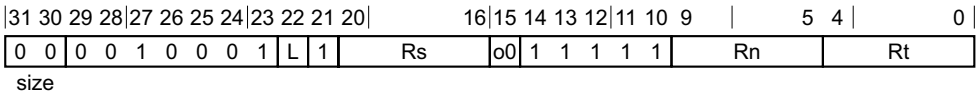
For information about memory accesses, see *Load/store addressing modes*.

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

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

No offset

(FEAT_LSE)



CASAB variant

Applies when L == 1 && o0 == 0.

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

CASALB variant

Applies when L == 1 && o0 == 1.

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

CASB variant

Applies when L == 0 && o0 == 0.

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

CASLB variant

Applies when L == 0 && o0 == 1.

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

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LSE) then UNDEFINED;

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

boolean acquire = L == '1';
```

```
boolean release = o0 == '1';
boolean tagchecked = n != 31;
```

Assembler symbols

<Ws>	Is the 32-bit name of the general-purpose register to be compared and loaded, encoded in the "Rs" field.
<Wt>	Is the 32-bit name of the general-purpose register to be conditionally stored, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(8) comparevalue;
bits(8) newvalue;
bits(8) data;

AccessDescriptor accdesc = CreateAccDescAtomicOp(MemAtomicOp_CAS, acquire, release, tagchecked);

comparevalue = X[s, 8];
newvalue = X[t, 8];

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

data = MemAtomic(address, comparevalue, newvalue, accdesc);

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

C6.2.45 CASH, CASAH, CASALH, CASLH

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

- CASAH and CASALH load from memory with acquire semantics.
- CASLH and CASALH store to memory with release semantics.
- CASH has neither acquire nor release semantics.

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

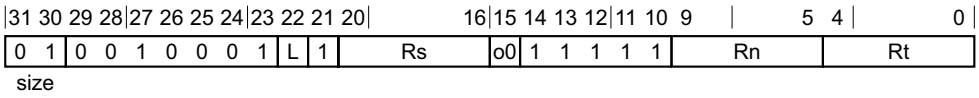
For information about memory accesses, see *Load/store addressing modes*.

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

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

No offset

(FEAT_LSE)



CASAH variant

Applies when L == 1 && o0 == 0.

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

CASALH variant

Applies when L == 1 && o0 == 1.

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

CASH variant

Applies when L == 0 && o0 == 0.

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

CASLH variant

Applies when L == 0 && o0 == 1.

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

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LSE) then UNDEFINED;

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

boolean acquire = L == '1';
```

```
boolean release = o0 == '1';
boolean tagchecked = n != 31;
```

Assembler symbols

<Ws> Is the 32-bit name of the general-purpose register to be compared and loaded, encoded in the "Rs" field.

<Wt> Is the 32-bit name of the general-purpose register to be conditionally stored, encoded in the "Rt" field.

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

Operation

```
bits(64) address;
bits(16) comparevalue;
bits(16) newvalue;
bits(16) data;

AccessDescriptor accdesc = CreateAccDescAtomicOp(MemAtomicOp_CAS, acquire, release, tagchecked);

comparevalue = X[s, 16];
newvalue = X[t, 16];

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

data = MemAtomic(address, comparevalue, newvalue, accdesc);

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

C6.2.46 CASP, CASPA, CASPAL, CASPL

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

- `CASPA` and `CASPAL` load from memory with acquire semantics.
- `CASPL` and `CASPAL` store to memory with release semantics.
- `CASP` has neither acquire nor release semantics.

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

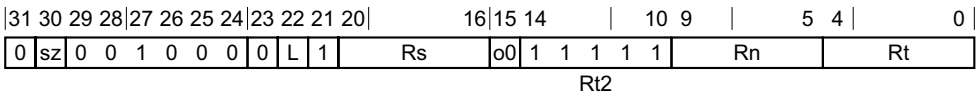
For information about memory accesses, see *Load/store addressing modes*.

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

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

No offset

(FEAT_LSE)



32-bit CASP variant

Applies when `sz == 0 && L == 0 && o0 == 0`.

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

32-bit CASPA variant

Applies when `sz == 0 && L == 1 && o0 == 0`.

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

32-bit CASPAL variant

Applies when `sz == 0 && L == 1 && o0 == 1`.

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

32-bit CASPL variant

Applies when `sz == 0 && L == 0 && o0 == 1`.

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

64-bit CASP variant

Applies when `sz == 1 && L == 0 && o0 == 0`.

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

64-bit CASPA variant

Applies when `sz == 1 && L == 1 && o0 == 0`.

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

64-bit CASPAL variant

Applies when sz == 1 && L == 1 && o0 == 1.

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

64-bit CASPL variant

Applies when sz == 1 && L == 0 && o0 == 1.

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

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LSE) then UNDEFINED;
if Rs<0> == '1' then UNDEFINED;
if Rt<0> == '1' then UNDEFINED;

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

constant integer datasize = 32 << UInt(sz);
boolean acquire = L == '1';
boolean release = o0 == '1';
boolean tagchecked = n != 31;
```

Assembler symbols

<Ws>	Is the 32-bit name of the first general-purpose register to be compared and loaded, encoded in the "Rs" field. <Ws> must be an even-numbered register.
<W(s+1)>	Is the 32-bit name of the second general-purpose register to be compared and loaded.
<Wt>	Is the 32-bit name of the first general-purpose register to be conditionally stored, encoded in the "Rt" field. <Wt> must be an even-numbered register.
<W(t+1)>	Is the 32-bit name of the second general-purpose register to be conditionally stored.
<Xs>	Is the 64-bit name of the first general-purpose register to be compared and loaded, encoded in the "Rs" field. <Xs> must be an even-numbered register.
<X(s+1)>	Is the 64-bit name of the second general-purpose register to be compared and loaded.
<Xt>	Is the 64-bit name of the first general-purpose register to be conditionally stored, encoded in the "Rt" field. <Xt> must be an even-numbered register.
<X(t+1)>	Is the 64-bit name of the second general-purpose register to be conditionally stored.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

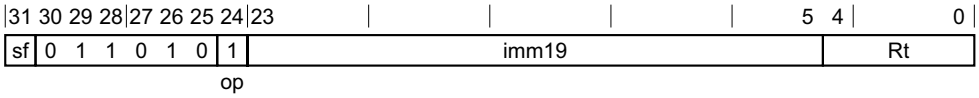
Operation

```
bits(64) address;
bits(2*datasize) comparevalue;
bits(2*datasize) newvalue;
bits(2*datasize) data;

bits(datasize) s1 = X[s, datasize];
bits(datasize) s2 = X[s+1, datasize];
bits(datasize) t1 = X[t, datasize];
bits(datasize) t2 = X[t+1, datasize];
```

```
AccessDescriptor accdesc = CreateAccDescAtomicOp(MemAtomicOp_CAS, acquire, release, tagchecked);

comparevalue = if BigEndian(accdesc.acctype) then s1:s2 else s2:s1;
newvalue = if BigEndian(accdesc.acctype) then t1:t2 else t2:t1;
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

data = MemAtomic(address, comparevalue, newvalue, accdesc);

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

C6.2.47 CBNZ

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



32-bit variant

Applies when `sf == 0`.

```
CBNZ <Wt>, <label>
```

64-bit variant

Applies when `sf == 1`.

```
CBNZ <Xt>, <label>
```

Decode for all variants of this encoding

```
integer t = UInt(Rt);
constant integer datasize = 32 << UInt(sf);
bits(64) offset = SignExtend(imm19:'00', 64);
```

Assembler symbols

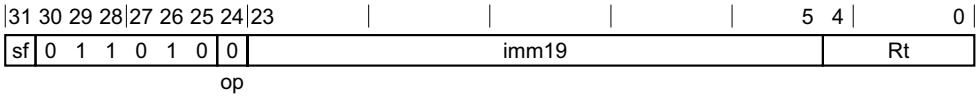
`<Wt>`	Is the 32-bit name of the general-purpose register to be tested, encoded in the "Rt" field.
`<Xt>`	Is the 64-bit name of the general-purpose register to be tested, encoded in the "Rt" field.
`<label>`	Is the program label to be conditionally branched to. Its offset from the address of this instruction, in the range +/-1MB, is encoded as "imm19" times 4.

Operation

```
bits(datasize) operand1 = X[t, datasize];
if IsZero(operand1) == FALSE then
    BranchTo(PC64 + offset, BranchType_DIR, TRUE);
else
    BranchNotTaken(BranchType_DIR, TRUE);
```

C6.2.48 CBZ

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



32-bit variant

Applies when `sf == 0`.

```
CBZ <Wt>, <label>
```

64-bit variant

Applies when `sf == 1`.

```
CBZ <Xt>, <label>
```

Decode for all variants of this encoding

```
integer t = UInt(Rt);
constant integer datasize = 32 << UInt(sf);
bits(64) offset = SignExtend(imm19:'00', 64);
```

Assembler symbols

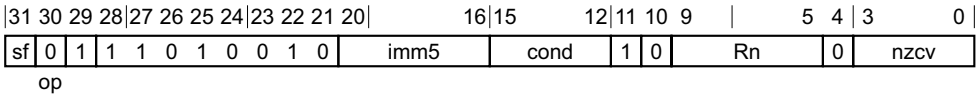
<Wt>	Is the 32-bit name of the general-purpose register to be tested, encoded in the "Rt" field.
<Xt>	Is the 64-bit name of the general-purpose register to be tested, encoded in the "Rt" field.
<label>	Is the program label to be conditionally branched to. Its offset from the address of this instruction, in the range +/-1MB, is encoded as "imm19" times 4.

Operation

```
bits(datasize) operand1 = X[t, datasize];
if IsZero(operand1) == TRUE then
    BranchTo(PC64 + offset, BranchType_DIR, TRUE);
else
    BranchNotTaken(BranchType_DIR, TRUE);
```

C6.2.49 CCMN (immediate)

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



32-bit variant

Applies when `sf == 0`.

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

64-bit variant

Applies when `sf == 1`.

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

Decode for all variants of this encoding

```
integer n = UInt(Rn);
constant integer datasize = 32 << UInt(sf);
bits(4) flags = nzcv;
bits(datasize) imm = ZeroExtend(imm5, datasize);
```

Assembler symbols

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

Operation

```
if ConditionHolds(cond) then
    bits(datasize) operand1 = X[n, datasize];
    (-, flags) = AddWithCarry(operand1, imm, '0');
PSTATE.<N,Z,C,V> = flags;
```

Operational information

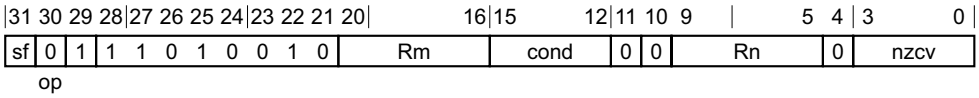
If PSTATE.DIT is 1:

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

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

C6.2.50 CCMN (register)

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



32-bit variant

Applies when `sf == 0`.

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

64-bit variant

Applies when `sf == 1`.

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

Decode for all variants of this encoding

```
integer n = UInt(Rn);
integer m = UInt(Rm);
constant integer datasize = 32 << UInt(sf);
bits(4) flags = nzcv;
```

Assembler symbols

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

Operation

```
if ConditionHolds(cond) then
    bits(datasize) operand1 = X[n, datasize];
    bits(datasize) operand2 = X[m, datasize];
    (-, flags) = AddWithCarry(operand1, operand2, '0');
PSTATE.<N,Z,C,V> = flags;
```

Operational information

If PSTATE.DIT is 1:

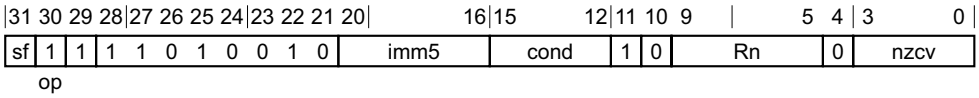
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.

  - The values of the NZCV flags.

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

C6.2.51 CCMP (immediate)

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



32-bit variant

Applies when `sf == 0`.

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

64-bit variant

Applies when `sf == 1`.

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

Decode for all variants of this encoding

```
integer n = UInt(Rn);
constant integer datasize = 32 << UInt(sf);
bits(4) flags = nzcv;
bits(datasize) imm = ZeroExtend(imm5, datasize);
```

Assembler symbols

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

Operation

```
if ConditionHolds(cond) then
    bits(datasize) operand1 = X[n, datasize];
    bits(datasize) operand2;
    operand2 = NOT(imm);
    (-, flags) = AddWithCarry(operand1, operand2, '1');
PSTATE.<N,Z,C,V> = flags;
```

Operational information

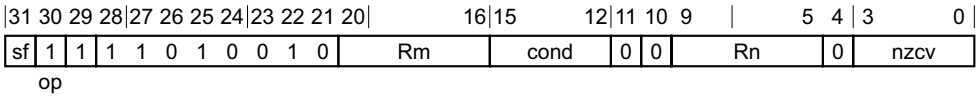
If PSTATE.DIT is 1:

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

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

C6.2.52 CCMP (register)

Conditional Compare (register) sets the value of the condition flags to the result of the comparison of two registers if the condition is TRUE, and an immediate value otherwise.



32-bit variant

Applies when `sf == 0`.

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

64-bit variant

Applies when `sf == 1`.

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

Decode for all variants of this encoding

```
integer n = UInt(Rn);
integer m = UInt(Rm);
constant integer datasize = 32 << UInt(sf);
bits(4) flags = nzcv;
```

Assembler symbols

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

Operation

```
if ConditionHolds(cond) then
    bits(datasize) operand1 = X[n, datasize];
    bits(datasize) operand2 = X[m, datasize];
    operand2 = NOT(operand2);
    (-, flags) = AddWithCarry(operand1, operand2, '1');
PSTATE.<N,Z,C,V> = flags;
```

Operational information

If PSTATE.DIT is 1:

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

  - The values of the NZCV flags.

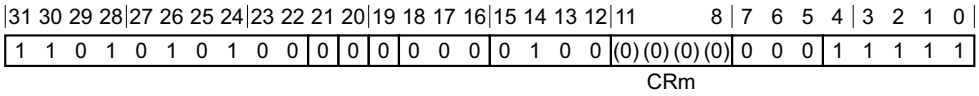
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

C6.2.53 CFINV

Invert Carry Flag. This instruction inverts the value of the PSTATE.C flag.

System

(FEAT_FlagM)



Encoding

CFINV

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_FlagM) then UNDEFINED;
```

Operation

```
PSTATE.C = NOT(PSTATE.C);
```

Operational information

If PSTATE.DIT is 1:

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

C6.2.54 CFP

Control Flow Prediction Restriction by Context prevents control flow predictions that predict execution addresses based on information gathered from earlier execution within a particular execution context. Control flow predictions determined by the actions of code in the target execution context or contexts appearing in program order before the instruction cannot be used to exploitatively control speculative execution occurring after the instruction is complete and synchronized.

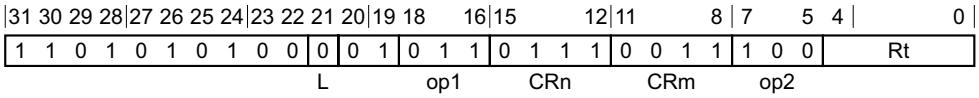
For more information, see CFP RCTX.

This instruction is an alias of the SYS instruction. This means that:

- The encodings in this description are named to match the encodings of SYS.
- The description of SYS gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

System

(FEAT_SPECRES)



Encoding

```
CFP RCTX, <Xt>
```

is equivalent to

```
SYS #3, C7, C3, #4, <Xt>
```

and is always the preferred disassembly.

Assembler symbols

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

Operation

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

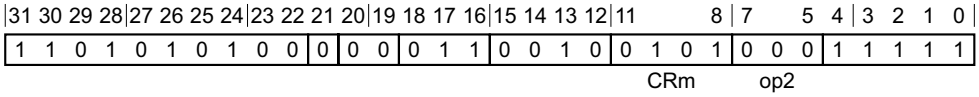
C6.2.55 CHKFEAT

Check feature status. This instruction indicates the status of features.

If FEAT_CHK is not implemented, this instruction executes as a NOP.

System

(FEAT_CHK)



Encoding

CHKFEAT X16

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_CHK) then EndOfInstruction();
```

Operation

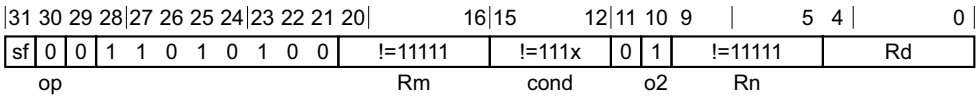
```
X[16, 64] = AArch64.ChkFeat(X[16, 64]);
```

C6.2.56 CINC

Conditional Increment returns, in the destination register, the value of the source register incremented by 1 if the condition is TRUE, and otherwise returns the value of the source register.

This instruction is an alias of the CSINC instruction. This means that:

- The encodings in this description are named to match the encodings of CSINC.
- The description of CSINC gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.



32-bit variant

Applies when `sf == 0`.

`CINC <Wd>, <Wn>, <cond>`

is equivalent to

`CSINC <Wd>, <Wn>, <Wn>, invert(<cond>)`

and is the preferred disassembly when `Rn == Rm`.

64-bit variant

Applies when `sf == 1`.

`CINC <Xd>, <Xn>, <cond>`

is equivalent to

`CSINC <Xd>, <Xn>, <Xn>, invert(<cond>)`

and is the preferred disassembly when `Rn == Rm`.

Assembler symbols

`<Wd>`	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
`<Wn>`	Is the 32-bit name of the general-purpose source register, encoded in the "Rn" and "Rm" fields.
`<Xd>`	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
`<Xn>`	Is the 64-bit name of the general-purpose source register, encoded in the "Rn" and "Rm" fields.
`<cond>`	Is one of the standard conditions, excluding AL and NV, encoded in the "cond" field with its least significant bit inverted.

Operation

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

Operational information

If PSTATE.DIT is 1:

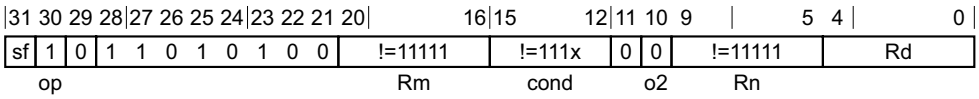
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.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

C6.2.57 CINV

Conditional Invert returns, in the destination register, the bitwise inversion of the value of the source register if the condition is TRUE, and otherwise returns the value of the source register.

This instruction is an alias of the CSINV instruction. This means that:

- The encodings in this description are named to match the encodings of CSINV.
- The description of CSINV gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.



32-bit variant

Applies when `sf == 0`.

`CINV <Wd>, <Wn>, <cond>`

is equivalent to

`CSINV <Wd>, <Wn>, <Wn>, invert(<cond>)`

and is the preferred disassembly when `Rn == Rm`.

64-bit variant

Applies when `sf == 1`.

`CINV <Xd>, <Xn>, <cond>`

is equivalent to

`CSINV <Xd>, <Xn>, <Xn>, invert(<cond>)`

and is the preferred disassembly when `Rn == Rm`.

Assembler symbols

`<Wd>`	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
`<Wn>`	Is the 32-bit name of the general-purpose source register, encoded in the "Rn" and "Rm" fields.
`<Xd>`	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
`<Xn>`	Is the 64-bit name of the general-purpose source register, encoded in the "Rn" and "Rm" fields.
`<cond>`	Is one of the standard conditions, excluding AL and NV, encoded in the "cond" field with its least significant bit inverted.

Operation

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

Operational information

If PSTATE.DIT is 1:

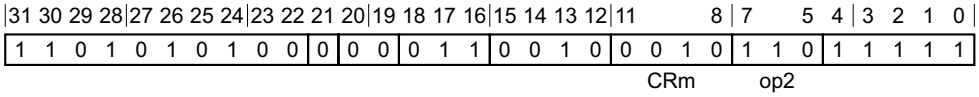
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.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

C6.2.58 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.

System

(FEAT_CLRBHB)



Encoding

CLRBHB

Decode for this encoding

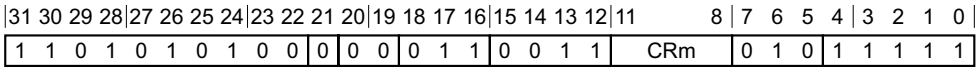
```
if !IsFeatureImplemented(FEAT_CLRBHB) then
    EndOfInstruction();
```

Operation

```
Hint_CLRBHB();
```

C6.2.59 CLREX

Clear Exclusive clears the local monitor of the executing PE.



Encoding

```
CLREX {#<imm>}
```

Decode for this encoding

```
// CRm field is ignored
```

Assembler symbols

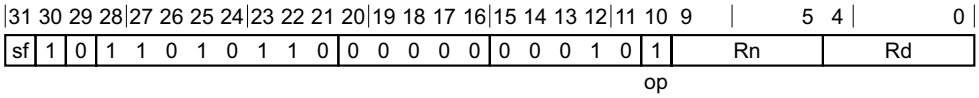
<imm> Is an optional 4-bit unsigned immediate, in the range 0 to 15, defaulting to 15 and encoded in the "CRm" field.

Operation

```
ClearExclusiveLocal(ProcessorID());
```

C6.2.60 CLS

Count Leading Sign bits counts the number of leading bits of the source register that have the same value as the most significant bit of the register, and writes the result to the destination register. This count does not include the most significant bit of the source register.



32-bit variant

Applies when `sf == 0`.

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

64-bit variant

Applies when `sf == 1`.

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

Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
constant integer datasize = 32 << UInt(sf);
```

Assembler symbols

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

Operation

```
integer result;
bits(datasize) operand1 = X[n, datasize];

result = CountLeadingSignBits(operand1);

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

Operational information

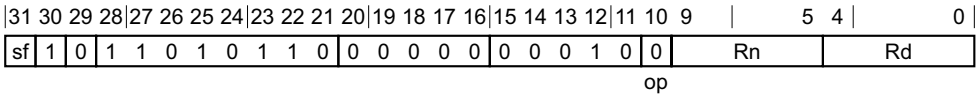
If PSTATE.DIT is 1:

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

— The values of the NZCV flags.

C6.2.61 CLZ

Count Leading Zeros counts the number of consecutive binary zero bits, starting from the most significant bit in the source register, and places the count in the destination register.



32-bit variant

Applies when `sf == 0`.

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

64-bit variant

Applies when `sf == 1`.

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

Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
constant integer datasize = 32 << UInt(sf);
```

Assembler symbols

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

Operation

```
integer result;
bits(datasize) operand1 = X[n, datasize];

result = CountLeadingZeroBits(operand1);
X[d, datasize] = result<datasize-1:0>;
```

Operational information

If PSTATE.DIT is 1:

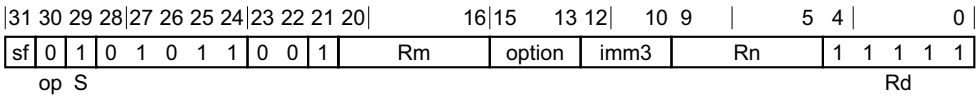
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.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

C6.2.62 CMN (extended register)

Compare Negative (extended register) adds a register value and a sign or zero-extended register value, followed by an optional left shift amount. The argument that is extended from the <Rm> register can be a byte, halfword, word, or doubleword. It updates the condition flags based on the result, and discards the result.

This instruction is an alias of the ADDS (extended register) instruction. This means that:

- The encodings in this description are named to match the encodings of ADDS (extended register).
- The description of ADDS (extended register) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.



32-bit variant

Applies when `sf == 0`.

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

is equivalent to

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

and is always the preferred disassembly.

64-bit variant

Applies when `sf == 1`.

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

is equivalent to

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

and is always the preferred disassembly.

Assembler symbols

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

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

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

<R> Is a width specifier, encoded in the "option" field. It can have the following values:

W	when `option = 00x`
W	when `option = 010`
X	when `option = x11`
W	when `option = 10x`
W	when `option = 110`

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

<extend> For the 32-bit variant: is the extension to be applied to the second source operand, encoded in the "option" field. It can have the following values:

UXTB when option = 000

UXTH when option = 001

LSL|UXTW when option = 010

UXTX when option = 011

SXTB when option = 100

SXTH when option = 101

SXTW when option = 110

SXTX when option = 111

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

For the 64-bit variant: is the extension to be applied to the second source operand, encoded in the "option" field. It can have the following values:

UXTB when option = 000

UXTH when option = 001

UXTW when option = 010

LSL|UXTX when option = 011

SXTB when option = 100

SXTH when option = 101

SXTW when option = 110

SXTX when option = 111

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

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

Operation

The description of ADDS (extended register) gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

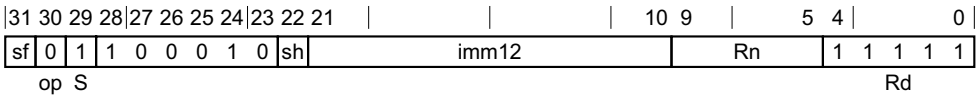
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.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

C6.2.63 CMN (immediate)

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

This instruction is an alias of the ADDS (immediate) instruction. This means that:

- The encodings in this description are named to match the encodings of ADDS (immediate).
- The description of ADDS (immediate) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.



32-bit variant

Applies when `sf == 0`.

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

is equivalent to

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

and is always the preferred disassembly.

64-bit variant

Applies when `sf == 1`.

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

is equivalent to

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

and is always the preferred disassembly.

Assembler symbols

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

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

`<imm>` Is an unsigned immediate, in the range 0 to 4095, encoded in the "imm12" field.

`<shift>` Is the optional left shift to apply to the immediate, defaulting to LSL #0 and encoded in the "sh" field. It can have the following values:

`LSL #0` when `sh = 0`

`LSL #12` when `sh = 1`

Operation

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

Operational information

If PSTATE.DIT is 1:

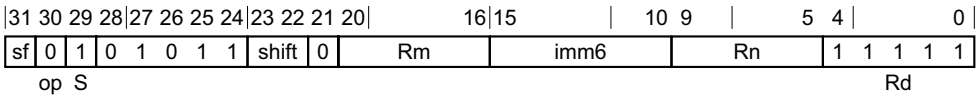
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.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

C6.2.64 CMN (shifted register)

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

This instruction is an alias of the ADDS (shifted register) instruction. This means that:

- The encodings in this description are named to match the encodings of ADDS (shifted register).
- The description of ADDS (shifted register) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.



32-bit variant

Applies when `sf == 0`.

```
CMN <Wn>, <Wm>{, <shift> #<amount>}
```

is equivalent to

```
ADDS WZR, <Wn>, <Wm> {, <shift> #<amount>}
```

and is always the preferred disassembly.

64-bit variant

Applies when `sf == 1`.

```
CMN <Xn>, <Xm>{, <shift> #<amount>}
```

is equivalent to

```
ADDS XZR, <Xn>, <Xm> {, <shift> #<amount>}
```

and is always the preferred disassembly.

Assembler symbols

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

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

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

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

`<shift>` Is the optional shift type to be applied to the second source operand, defaulting to LSL and encoded in the "shift" field. It can have the following values:

`LSL` when `shift = 00`

`LSR` when `shift = 01`

`ASR` when `shift = 10`

The encoding `shift = 11` is reserved.

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

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

Operation

The description of ADDS (shifted register) gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

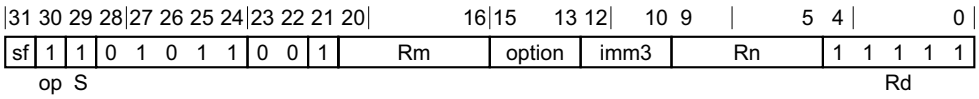
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.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

C6.2.65 CMP (extended register)

Compare (extended register) subtracts a sign or zero-extended register value, followed by an optional left shift amount, from a register value. The argument that is extended from the <Rm> register can be a byte, halfword, word, or doubleword. It updates the condition flags based on the result, and discards the result.

This instruction is an alias of the SUBS (extended register) instruction. This means that:

- The encodings in this description are named to match the encodings of SUBS (extended register).
- The description of SUBS (extended register) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.



32-bit variant

Applies when `sf == 0`.

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

is equivalent to

```
SUBS WZR, <Wn|WSP>, <Wm>{, <extend> {#<amount>}}
```

and is always the preferred disassembly.

64-bit variant

Applies when `sf == 1`.

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

is equivalent to

```
SUBS XZR, <Xn|SP>, <R><m>{, <extend> {#<amount>}}
```

and is always the preferred disassembly.

Assembler symbols

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

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

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

<R> Is a width specifier, encoded in the "option" field. It can have the following values:

W	when `option = 00x`
W	when `option = 010`
X	when `option = x11`
W	when `option = 10x`
W	when `option = 110`

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

<extend> For the 32-bit variant: is the extension to be applied to the second source operand, encoded in the "option" field. It can have the following values:

UXTB when option = 000

UXTH when option = 001

LSL|UXTW when option = 010

UXTX when option = 011

SXTB when option = 100

SXTH when option = 101

SXTW when option = 110

SXTX when option = 111

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

For the 64-bit variant: is the extension to be applied to the second source operand, encoded in the "option" field. It can have the following values:

UXTB when option = 000

UXTH when option = 001

UXTW when option = 010

LSL|UXTX when option = 011

SXTB when option = 100

SXTH when option = 101

SXTW when option = 110

SXTX when option = 111

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

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

Operation

The description of SUBS (extended register) gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

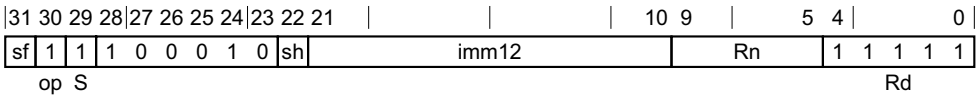
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.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

C6.2.66 CMP (immediate)

Compare (immediate) subtracts an optionally-shifted immediate value from a register value. It updates the condition flags based on the result, and discards the result.

This instruction is an alias of the SUBS (immediate) instruction. This means that:

- The encodings in this description are named to match the encodings of SUBS (immediate).
- The description of SUBS (immediate) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.



32-bit variant

Applies when `sf == 0`.

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

is equivalent to

```
SUBS WZR, <Wn|WSP>, #<imm> {, <shift>}
```

and is always the preferred disassembly.

64-bit variant

Applies when `sf == 1`.

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

is equivalent to

```
SUBS XZR, <Xn|SP>, #<imm> {, <shift>}
```

and is always the preferred disassembly.

Assembler symbols

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

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

`<imm>` Is an unsigned immediate, in the range 0 to 4095, encoded in the "imm12" field.

`<shift>` Is the optional left shift to apply to the immediate, defaulting to LSL #0 and encoded in the "sh" field. It can have the following values:

`LSL #0` when `sh = 0`

`LSL #12` when `sh = 1`

Operation

The description of SUBS (immediate) gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

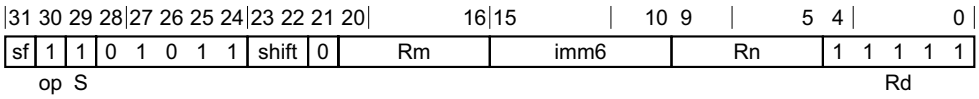
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.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

C6.2.67 CMP (shifted register)

Compare (shifted register) subtracts an optionally-shifted register value from a register value. It updates the condition flags based on the result, and discards the result.

This instruction is an alias of the SUBS (shifted register) instruction. This means that:

- The encodings in this description are named to match the encodings of SUBS (shifted register).
- The description of SUBS (shifted register) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.



32-bit variant

Applies when `sf == 0`.

```
CMP <Wn>, <Wm>{, <shift> #<amount>}
```

is equivalent to

```
SUBS WZR, <Wn>, <Wm> {, <shift> #<amount>}
```

and is always the preferred disassembly.

64-bit variant

Applies when `sf == 1`.

```
CMP <Xn>, <Xm>{, <shift> #<amount>}
```

is equivalent to

```
SUBS XZR, <Xn>, <Xm> {, <shift> #<amount>}
```

and is always the preferred disassembly.

Assembler symbols

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

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

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

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

`<shift>` Is the optional shift type to be applied to the second source operand, defaulting to LSL and encoded in the "shift" field. It can have the following values:

- `LSL` when `shift = 00`
- `LSR` when `shift = 01`
- `ASR` when `shift = 10`

The encoding `shift = 11` is reserved.

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

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

Operation

The description of SUBS (shifted register) gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

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

C6.2.68 CMPP

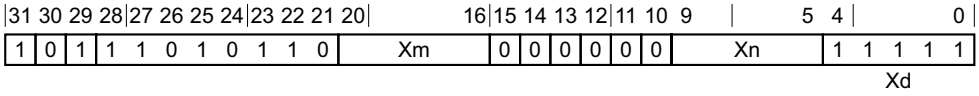
Compare with Tag subtracts the 56-bit address held in the second source register from the 56-bit address held in the first source register, updates the condition flags based on the result of the subtraction, and discards the result.

This instruction is an alias of the SUBPS instruction. This means that:

- The encodings in this description are named to match the encodings of SUBPS.
- The description of SUBPS gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Integer

(FEAT_MTE)



Encoding

`CMPP <Xn|SP>, <Xm|SP>`

is equivalent to

`SUBPS XZR, <Xn|SP>, <Xm|SP>`

and is always the preferred disassembly.

Assembler symbols

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

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

Operation

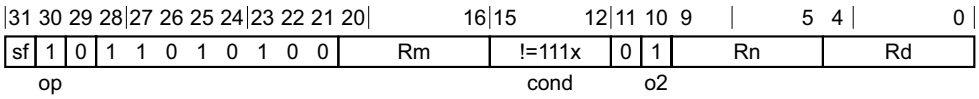
The description of SUBPS gives the operational pseudocode for this instruction.

C6.2.69 CNEG

Conditional Negate returns, in the destination register, the negated value of the source register if the condition is TRUE, and otherwise returns the value of the source register.

This instruction is an alias of the CSNEG instruction. This means that:

- The encodings in this description are named to match the encodings of CSNEG.
- The description of CSNEG gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.



32-bit variant

Applies when `sf == 0`.

`CNEG <Wd>, <Wn>, <cond>`

is equivalent to

`CSNEG <Wd>, <Wn>, <Wn>, invert(<cond>)`

and is the preferred disassembly when `Rn == Rm`.

64-bit variant

Applies when `sf == 1`.

`CNEG <Xd>, <Xn>, <cond>`

is equivalent to

`CSNEG <Xd>, <Xn>, <Xn>, invert(<cond>)`

and is the preferred disassembly when `Rn == Rm`.

Assembler symbols

`<Wd>`	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
`<Wn>`	Is the 32-bit name of the general-purpose source register, encoded in the "Rn" and "Rm" fields.
`<Xd>`	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
`<Xn>`	Is the 64-bit name of the general-purpose source register, encoded in the "Rn" and "Rm" fields.
`<cond>`	Is one of the standard conditions, excluding AL and NV, encoded in the "cond" field with its least significant bit inverted.

Operation

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

Operational information

If PSTATE.DIT is 1:

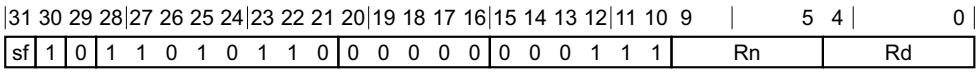
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.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

C6.2.70 CNT

Count bits counts the number of binary one bits in the value of the source register, and writes the result to the destination register.

Integer

(FEAT_CSSC)



32-bit variant

Applies when `sf == 0`.

```
CNT <Wd>, <Wn>
```

64-bit variant

Applies when `sf == 1`.

```
CNT <Xd>, <Xn>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_CSSC) then UNDEFINED;
constant integer datasize = 32 << UInt(sf);
integer n = UInt(Rn);
integer d = UInt(Rd);
```

Assembler symbols

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

Operation

```
bits(datasize) operand1 = X[n, datasize];
integer result = BitCount(operand1);
X[d, datasize] = result<datasize-1:0>;
```

Operational information

If PSTATE.DIT is 1:

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

— The values of the NZCV flags.

C6.2.71 COSP

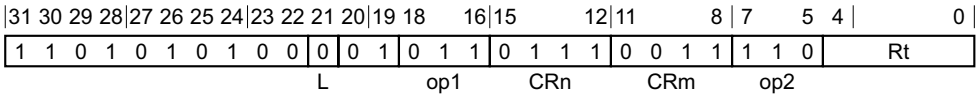
Clear Other Speculative Prediction Restriction by Context prevents predictions, other than Cache prefetch, Control flow, and Data Value predictions, that predict execution addresses based on information gathered from earlier execution within a particular execution context. Predictions, other than Cache prefetch, Control flow, and Data Value predictions, determined by the actions of code in the target execution context or contexts appearing in program order before the instruction cannot exploitatively control any speculative access occurring after the instruction is complete and synchronized.

This instruction is an alias of the SYS instruction. This means that:

- The encodings in this description are named to match the encodings of SYS.
- The description of SYS gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

System

(FEAT_SPECRES2)



Encoding

```
COSP RCTX, <Xt>
```

is equivalent to

```
SYS #3, C7, C3, #6, <Xt>
```

and is always the preferred disassembly.

Assembler symbols

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

Operation

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

C6.2.72 CPP

Cache Prefetch Prediction Restriction by Context prevents cache allocation predictions that predict execution addresses based on information gathered from earlier execution within a particular execution context. The actions of code in the target execution context or contexts appearing in program order before the instruction cannot exploitatively control cache prefetch predictions occurring after the instruction is complete and synchronized.

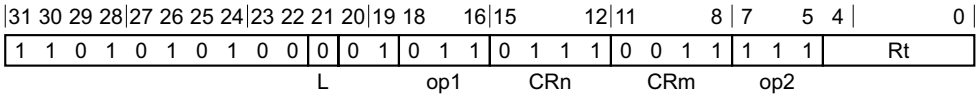
For more information, see CPP RCTX.

This instruction is an alias of the SYS instruction. This means that:

- The encodings in this description are named to match the encodings of SYS.
- The description of SYS gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

System

(FEAT_SPECRES)



Encoding

`CPP RCTX, <Xt>`

is equivalent to

`SYS #3, C7, C3, #7, <Xt>`

and is always the preferred disassembly.

Assembler symbols

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

Operation

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

C6.2.73 CPYFP, CPYFM, CPYFE

Memory Copy Forward-only. These instructions perform a memory copy. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYFP, then CPYFM, and then CPYFE.

CPYFP performs some preconditioning of the arguments suitable for using the CPYFM instruction, and performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYFM performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYFE performs the last part of the memory copy.

Note

The inclusion of IMPLEMENTATION DEFINED amounts of memory copy allows some optimization of the size that can be performed.

The memory copy performed by these instructions is in the forward direction only, so the instructions are suitable for a memory copy only where there is no overlap between the source and destination locations, or where the source address is greater than the destination address.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

After execution of CPYFP, option A (which results in encoding PSTATE.C = 0):

- If Xn<63> == 1, the copy size is saturated to `0x7FFFFFFFFFFFFFFF`.
- Xs holds the original Xs + saturated Xn.
- Xd holds the original Xd + saturated Xn.
- Xn holds -1* saturated Xn + an IMPLEMENTATION DEFINED number of bytes copied.
- PSTATE.{N,Z,V} are set to {0,0,0}.

After execution of CPYFP, option B (which results in encoding PSTATE.C = 1):

- If Xn<63> == 1, the copy size is saturated to `0x7FFFFFFFFFFFFFFF`.
- Xs holds the original Xs + an IMPLEMENTATION DEFINED number of bytes copied.
- Xd holds the original Xd + an IMPLEMENTATION DEFINED number of bytes copied.
- Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
- PSTATE.{N,Z,V} are set to {0,0,0}.

For CPYFM, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number and holds -1* the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from -Xn.
- Xd holds the lowest address that the copy is made to -Xn.
- At the end of the instruction, the value of Xn is written back with -1* the number of bytes remaining to be copied in the memory copy in total.

For CPYFM, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from.

- Xd holds the lowest address that the copy is copied to.
- At the end of the instruction:
  - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
  - the value of Xs is written back with the lowest address that has not been copied from.
  - the value of Xd is written back with the lowest address that has not been copied to.

For CPYFE, option A (encoded by PSTATE.C = 0), the format of the arguments is:

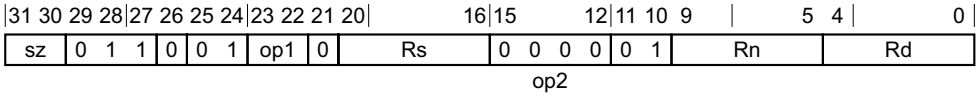
- Xn is treated as a signed 64-bit number and holds -1* the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from -Xn.
- Xd holds the lowest address that the copy is made to -Xn.
- At the end of the instruction, the value of Xn is written back with 0.

For CPYFE, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from.
- Xd holds the lowest address that the copy is copied to.
- At the end of the instruction:
  - the value of Xn is written back with 0.
  - the value of Xs is written back with the lowest address that has not been copied from.
  - the value of Xd is written back with the lowest address that has not been copied to.

Integer

(FEAT_MOPS)



Epilogue variant

Applies when `op1 == 10`.

```
CPYFE [<Xd>]!, [<Xs>]!, <Xn>!
```

Main variant

Applies when `op1 == 01`.

```
CPYFM [<Xd>]!, [<Xs>]!, <Xn>!
```

Prologue variant

Applies when `op1 == 00`.

```
CPYFP [<Xd>]!, [<Xs>]!, <Xn>!
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_MOPS) || sz != '00' then UNDEFINED;

CPYParams memcpy;
memcpy.d = UInt(Rd);
memcpy.s = UInt(Rs);
memcpy.n = UInt(Rn);
bits(4) options = op2;
boolean rnontemporal = options<3> == '1';
boolean wnontemporal = options<2> == '1';

case op1 of
    when '00' memcpy.stage = MOPSStage_Prologue;
    when '01' memcpy.stage = MOPSStage_Main;
    when '10' memcpy.stage = MOPSStage_Epilogue;
    otherwise SEE "Memory Copy and Memory Set";

CheckMOPSEnabled();

if (memcpy.s == memcpy.n || memcpy.s == memcpy.d || memcpy.n == memcpy.d || memcpy.d == 31 || memcpy.s
== 31 || memcpy.n == 31) then
    Constraint c = ConstrainUnpredictable(Unpredictable_MOPSOVERLAP31);
    assert c IN {Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNDEF UNDEFINED;
        when Constraint_NOP   EndOfInstruction();
```

Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*, and particularly *Memory Copy and Memory Set CPY**.

Assembler symbols

<Xd> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the "Rd" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.

<Xs> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the "Rs" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the "Rs" field.

<Xn> For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero at the end of the instruction, encoded in the "Rn" field.

For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the "Rn" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the "Rn" field.

Operation

```
constant integer N = MaxBlockSizeCopiedBytes();
bits(8*N) readdata;

memcpy.nzcv = PSTATE.<N,Z,C,V>;
memcpy.toaddress = X[memcpy.d, 64];
```

```
memcpy.fromaddress = X[memcpy.s, 64];
memcpy.cpysize = SInt(X[memcpy.n, 64]);
memcpy.implements_option_a = CPYFOptionA();

boolean rprivileged = if options<1> == '1' then AArch64.IsUnprivAccessPriv() else PSTATE.EL != EL0;
boolean wprivileged = if options<0> == '1' then AArch64.IsUnprivAccessPriv() else PSTATE.EL != EL0;

AccessDescriptor raccdesc = CreateAccDescMOPS(MemOp_LOAD, rprivileged, rnontemporal);
AccessDescriptor waccdesc = CreateAccDescMOPS(MemOp_STORE, wprivileged, wnontemporal);

if memcpy.stage == MOPSStage_Prologue then
    if memcpy.cpysize<63> == '1' then memcpy.cpysize = 0x7FFFFFFFFFFFFFFF;

    if memcpy.implements_option_a then
        memcpy.nzcv = '0000';
        // Copy in the forward direction offsets the arguments.
        memcpy.toaddress = memcpy.toaddress + memcpy.cpysize;
        memcpy.fromaddress = memcpy.fromaddress + memcpy.cpysize;
        memcpy.cpysize = 0 - memcpy.cpysize;
    else
        memcpy.nzcv = '0010';

memcpy.stagecpysize = MemCpyStageSize(memcpy);

if memcpy.stage != MOPSStage_Prologue then
    CheckMemCpyParams(memcpy, options);

integer copied;
boolean iswrite;
AddressDescriptor memaddrdesc;
PhysMemRetStatus memstatus;
memcpy.forward = TRUE;
boolean fault = FALSE;
integer B;

if memcpy.implements_option_a then
    while memcpy.stagecpysize != 0 && !fault do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(memcpy);
        assert B <= -1 * memcpy.stagecpysize;

        (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress + memcpy.cpysize,
memcpy.fromaddress + memcpy.cpysize, memcpy.forward, B, raccdesc, waccdesc);
        if copied != B then
            fault = TRUE;
        else
            memcpy.cpysize = memcpy.cpysize + B;
            memcpy.stagecpysize = memcpy.stagecpysize + B;
else
    while memcpy.stagecpysize > 0 && !fault do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(memcpy);
        assert B <= memcpy.stagecpysize;

        (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress, memcpy.fromaddress,
memcpy.forward, B, raccdesc, waccdesc);
        if copied != B then
            fault = TRUE;
        else
            memcpy.fromaddress = memcpy.fromaddress + B;
            memcpy.toaddress = memcpy.toaddress + B;
            memcpy.cpysize = memcpy.cpysize - B;
            memcpy.stagecpysize = memcpy.stagecpysize - B;

UpdateCpyRegisters(memcpy, fault, copied);
```

```
if fault then
    if IsFault(memaddrdesc) then
        AArch64.Abort(memaddrdesc.vaddress, memaddrdesc.fault);

    if IsFault(memstatus) then
        AccessDescriptor accdesc = if iswrite then waccdesc else raccdesc;
        HandleExternalAbort(memstatus, iswrite, memaddrdesc, B, accdesc);

if memcpy.stage == MOPSStage_Prologue then
    PSTATE.<N,Z,C,V> = memcpy.nzcv;
```

C6.2.74 CPYFPN, CPYFMN, CPYFEN

Memory Copy Forward-only, reads and writes non-temporal. These instructions perform a memory copy. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYFPN, then CPYFMN, and then CPYFEN.

CPYFPN performs some preconditioning of the arguments suitable for using the CPYFMN instruction, and performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYFMN performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYFEN performs the last part of the memory copy.

Note

The inclusion of IMPLEMENTATION DEFINED amounts of memory copy allows some optimization of the size that can be performed.

The memory copy performed by these instructions is in the forward direction only, so the instructions are suitable for a memory copy only where there is no overlap between the source and destination locations, or where the source address is greater than the destination address.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

After execution of CPYFPN, option A (which results in encoding PSTATE.C = 0):

- If Xn<63> == 1, the copy size is saturated to `0x7FFFFFFFFFFFFFFF`.
- Xs holds the original Xs + saturated Xn.
- Xd holds the original Xd + saturated Xn.
- Xn holds -1* saturated Xn + an IMPLEMENTATION DEFINED number of bytes copied.
- PSTATE.{N,Z,V} are set to {0,0,0}.

After execution of CPYFPN, option B (which results in encoding PSTATE.C = 1):

- If Xn<63> == 1, the copy size is saturated to `0x7FFFFFFFFFFFFFFF`.
- Xs holds the original Xs + an IMPLEMENTATION DEFINED number of bytes copied.
- Xd holds the original Xd + an IMPLEMENTATION DEFINED number of bytes copied.
- Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
- PSTATE.{N,Z,V} are set to {0,0,0}.

For CPYFMN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number and holds -1* the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from -Xn.
- Xd holds the lowest address that the copy is made to -Xn.
- At the end of the instruction, the value of Xn is written back with -1* the number of bytes remaining to be copied in the memory copy in total.

For CPYFMN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from.

- Xd holds the lowest address that the copy is copied to.
- At the end of the instruction:
  - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
  - the value of Xs is written back with the lowest address that has not been copied from.
  - the value of Xd is written back with the lowest address that has not been copied to.

For CPYFEN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

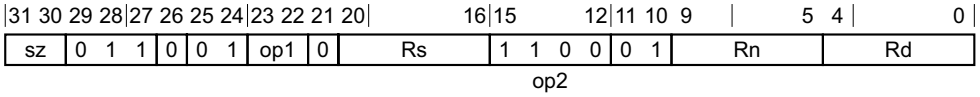
- Xn is treated as a signed 64-bit number and holds -1* the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from -Xn.
- Xd holds the lowest address that the copy is made to -Xn.
- At the end of the instruction, the value of Xn is written back with 0.

For CPYFEN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from.
- Xd holds the lowest address that the copy is copied to.
- At the end of the instruction:
  - the value of Xn is written back with 0.
  - the value of Xs is written back with the lowest address that has not been copied from.
  - the value of Xd is written back with the lowest address that has not been copied to.

Integer

(FEAT_MOPS)



Epilogue variant

Applies when `op1 == 10`.

```
CPYFEN [<Xd>]!, [<Xs>]!, <Xn>!
```

Main variant

Applies when `op1 == 01`.

```
CPYFMN [<Xd>]!, [<Xs>]!, <Xn>!
```

Prologue variant

Applies when `op1 == 00`.

```
CPYFPN [<Xd>]!, [<Xs>]!, <Xn>!
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_MOPS) || sz != '00' then UNDEFINED;

CPYParams memcpy;
memcpy.d = UInt(Rd);
memcpy.s = UInt(Rs);
memcpy.n = UInt(Rn);
bits(4) options = op2;
boolean rnontemporal = options<3> == '1';
boolean wnontemporal = options<2> == '1';

case op1 of
    when '00' memcpy.stage = MOPSStage_Prologue;
    when '01' memcpy.stage = MOPSStage_Main;
    when '10' memcpy.stage = MOPSStage_Epilogue;
    otherwise SEE "Memory Copy and Memory Set";

CheckMOPSEnabled();

if (memcpy.s == memcpy.n || memcpy.s == memcpy.d || memcpy.n == memcpy.d || memcpy.d == 31 || memcpy.s
== 31 || memcpy.n == 31) then
    Constraint c = ConstrainUnpredictable(Unpredictable_MOPSOVERLAP31);
    assert c IN {Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNDEF UNDEFINED;
        when Constraint_NOP   EndOfInstruction();
```

Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*, and particularly *Memory Copy and Memory Set CPY**.

Assembler symbols

<Xd> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the "Rd" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.

<Xs> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the "Rs" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the "Rs" field.

<Xn> For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero at the end of the instruction, encoded in the "Rn" field.

For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the "Rn" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the "Rn" field.

Operation

```
constant integer N = MaxBlockSizeCopiedBytes();
bits(8*N) readdata;

memcpy.nzcv = PSTATE.<N,Z,C,V>;
memcpy.toaddress = X[memcpy.d, 64];
```

```
memcpy.fromaddress = X[memcpy.s, 64];
memcpy.cpysize = SInt(X[memcpy.n, 64]);
memcpy.implements_option_a = CPYFOptionA();

boolean rprivileged = if options<1> == '1' then AArch64.IsUnprivAccessPriv() else PSTATE.EL != EL0;
boolean wprivileged = if options<0> == '1' then AArch64.IsUnprivAccessPriv() else PSTATE.EL != EL0;

AccessDescriptor raccdesc = CreateAccDescMOPS(MemOp_LOAD, rprivileged, rnontemporal);
AccessDescriptor waccdesc = CreateAccDescMOPS(MemOp_STORE, wprivileged, wnontemporal);

if memcpy.stage == MOPSStage_Prologue then
    if memcpy.cpysize<63> == '1' then memcpy.cpysize = 0x7FFFFFFFFFFFFFFF;

    if memcpy.implements_option_a then
        memcpy.nzcv = '0000';
        // Copy in the forward direction offsets the arguments.
        memcpy.toaddress = memcpy.toaddress + memcpy.cpysize;
        memcpy.fromaddress = memcpy.fromaddress + memcpy.cpysize;
        memcpy.cpysize = 0 - memcpy.cpysize;
    else
        memcpy.nzcv = '0010';

memcpy.stagecpysize = MemCpyStageSize(memcpy);

if memcpy.stage != MOPSStage_Prologue then
    CheckMemCpyParams(memcpy, options);

integer copied;
boolean iswrite;
AddressDescriptor memaddrdesc;
PhysMemRetStatus memstatus;
memcpy.forward = TRUE;
boolean fault = FALSE;
integer B;

if memcpy.implements_option_a then
    while memcpy.stagecpysize != 0 && !fault do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(memcpy);
        assert B <= -1 * memcpy.stagecpysize;

        (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress + memcpy.cpysize,
memcpy.fromaddress + memcpy.cpysize, memcpy.forward, B, raccdesc, waccdesc);
        if copied != B then
            fault = TRUE;
        else
            memcpy.cpysize = memcpy.cpysize + B;
            memcpy.stagecpysize = memcpy.stagecpysize + B;
else
    while memcpy.stagecpysize > 0 && !fault do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(memcpy);
        assert B <= memcpy.stagecpysize;

        (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress, memcpy.fromaddress,
memcpy.forward, B, raccdesc, waccdesc);
        if copied != B then
            fault = TRUE;
        else
            memcpy.fromaddress = memcpy.fromaddress + B;
            memcpy.toaddress = memcpy.toaddress + B;
            memcpy.cpysize = memcpy.cpysize - B;
            memcpy.stagecpysize = memcpy.stagecpysize - B;

UpdateCpyRegisters(memcpy, fault, copied);
```

```
if fault then
    if IsFault(memaddrdesc) then
        AArch64.Abort(memaddrdesc.vaddress, memaddrdesc.fault);

    if IsFault(memstatus) then
        AccessDescriptor accdesc = if iswrite then waccdesc else raccdesc;
        HandleExternalAbort(memstatus, iswrite, memaddrdesc, B, accdesc);

if memcpy.stage == MOPSStage_Prologue then
    PSTATE.<N,Z,C,V> = memcpy.nzcv;
```

C6.2.75 CPYFPRN, CPYFMRN, CPYFERN

Memory Copy Forward-only, reads non-temporal. These instructions perform a memory copy. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYFPRN, then CPYFMRN, and then CPYFERN.

CPYFPRN performs some preconditioning of the arguments suitable for using the CPYFMRN instruction, and performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYFMRN performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYFERN performs the last part of the memory copy.

Note

The inclusion of IMPLEMENTATION DEFINED amounts of memory copy allows some optimization of the size that can be performed.

The memory copy performed by these instructions is in the forward direction only, so the instructions are suitable for a memory copy only where there is no overlap between the source and destination locations, or where the source address is greater than the destination address.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

After execution of CPYFPRN, option A (which results in encoding PSTATE.C = 0):

- If Xn<63> == 1, the copy size is saturated to `0x7FFFFFFFFFFFFFFF`.
- Xs holds the original Xs + saturated Xn.
- Xd holds the original Xd + saturated Xn.
- Xn holds -1* saturated Xn + an IMPLEMENTATION DEFINED number of bytes copied.
- PSTATE.{N,Z,V} are set to {0,0,0}.

After execution of CPYFPRN, option B (which results in encoding PSTATE.C = 1):

- If Xn<63> == 1, the copy size is saturated to `0x7FFFFFFFFFFFFFFF`.
- Xs holds the original Xs + an IMPLEMENTATION DEFINED number of bytes copied.
- Xd holds the original Xd + an IMPLEMENTATION DEFINED number of bytes copied.
- Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
- PSTATE.{N,Z,V} are set to {0,0,0}.

For CPYFMRN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number and holds -1* the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from -Xn.
- Xd holds the lowest address that the copy is made to -Xn.
- At the end of the instruction, the value of Xn is written back with -1* the number of bytes remaining to be copied in the memory copy in total.

For CPYFMRN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from.

- Xd holds the lowest address that the copy is copied to.
- At the end of the instruction:
  - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
  - the value of Xs is written back with the lowest address that has not been copied from.
  - the value of Xd is written back with the lowest address that has not been copied to.

For CPYFERN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

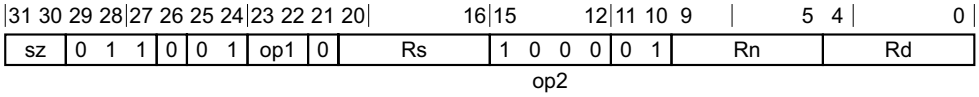
- Xn is treated as a signed 64-bit number and holds -1* the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from -Xn.
- Xd holds the lowest address that the copy is made to -Xn.
- At the end of the instruction, the value of Xn is written back with 0.

For CPYFERN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from.
- Xd holds the lowest address that the copy is copied to.
- At the end of the instruction:
  - the value of Xn is written back with 0.
  - the value of Xs is written back with the lowest address that has not been copied from.
  - the value of Xd is written back with the lowest address that has not been copied to.

Integer

(FEAT_MOPS)



Epilogue variant

Applies when `op1 == 10`.

```
CPYFERN [<Xd>]!, [<Xs>]!, <Xn>!
```

Main variant

Applies when `op1 == 01`.

```
CPYFMRN [<Xd>]!, [<Xs>]!, <Xn>!
```

Prologue variant

Applies when `op1 == 00`.

```
CPYFPRN [<Xd>]!, [<Xs>]!, <Xn>!
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_MOPS) || sz != '00' then UNDEFINED;

CPYParams memcpy;
memcpy.d = UInt(Rd);
memcpy.s = UInt(Rs);
memcpy.n = UInt(Rn);
bits(4) options = op2;
boolean rnontemporal = options<3> == '1';
boolean wnontemporal = options<2> == '1';

case op1 of
    when '00' memcpy.stage = MOPSStage_Prologue;
    when '01' memcpy.stage = MOPSStage_Main;
    when '10' memcpy.stage = MOPSStage_Epilogue;
    otherwise SEE "Memory Copy and Memory Set";

CheckMOPSEnabled();

if (memcpy.s == memcpy.n || memcpy.s == memcpy.d || memcpy.n == memcpy.d || memcpy.d == 31 || memcpy.s
== 31 || memcpy.n == 31) then
    Constraint c = ConstrainUnpredictable(Unpredictable_MOPSOVERLAP31);
    assert c IN {Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNDEF UNDEFINED;
        when Constraint_NOP   EndOfInstruction();
```

Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*, and particularly *Memory Copy and Memory Set CPY**.

Assembler symbols

<Xd> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the "Rd" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.

<Xs> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the "Rs" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the "Rs" field.

<Xn> For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero at the end of the instruction, encoded in the "Rn" field.

For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the "Rn" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the "Rn" field.

Operation

```
constant integer N = MaxBlockSizeCopiedBytes();
bits(8*N) readdata;

memcpy.nzcv = PSTATE.<N,Z,C,V>;
memcpy.toaddress = X[memcpy.d, 64];
```

```
memcpy.fromaddress = X[memcpy.s, 64];
memcpy.cpysize = SInt(X[memcpy.n, 64]);
memcpy.implements_option_a = CPYFOptionA();

boolean rprivileged = if options<1> == '1' then AArch64.IsUnprivAccessPriv() else PSTATE.EL != EL0;
boolean wprivileged = if options<0> == '1' then AArch64.IsUnprivAccessPriv() else PSTATE.EL != EL0;

AccessDescriptor raccdesc = CreateAccDescMOPS(MemOp_LOAD, rprivileged, rnontemporal);
AccessDescriptor waccdesc = CreateAccDescMOPS(MemOp_STORE, wprivileged, wnontemporal);

if memcpy.stage == MOPSStage_Prologue then
    if memcpy.cpysize<63> == '1' then memcpy.cpysize = 0x7FFFFFFFFFFFFFFF;

    if memcpy.implements_option_a then
        memcpy.nzcv = '0000';
        // Copy in the forward direction offsets the arguments.
        memcpy.toaddress = memcpy.toaddress + memcpy.cpysize;
        memcpy.fromaddress = memcpy.fromaddress + memcpy.cpysize;
        memcpy.cpysize = 0 - memcpy.cpysize;
    else
        memcpy.nzcv = '0010';

memcpy.stagecpysize = MemCpyStageSize(memcpy);

if memcpy.stage != MOPSStage_Prologue then
    CheckMemCpyParams(memcpy, options);

integer copied;
boolean iswrite;
AddressDescriptor memaddrdesc;
PhysMemRetStatus memstatus;
memcpy.forward = TRUE;
boolean fault = FALSE;
integer B;

if memcpy.implements_option_a then
    while memcpy.stagecpysize != 0 && !fault do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(memcpy);
        assert B <= -1 * memcpy.stagecpysize;

        (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress + memcpy.cpysize,
memcpy.fromaddress + memcpy.cpysize, memcpy.forward, B, raccdesc, waccdesc);
        if copied != B then
            fault = TRUE;
        else
            memcpy.cpysize = memcpy.cpysize + B;
            memcpy.stagecpysize = memcpy.stagecpysize + B;
else
    while memcpy.stagecpysize > 0 && !fault do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(memcpy);
        assert B <= memcpy.stagecpysize;

        (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress, memcpy.fromaddress,
memcpy.forward, B, raccdesc, waccdesc);
        if copied != B then
            fault = TRUE;
        else
            memcpy.fromaddress = memcpy.fromaddress + B;
            memcpy.toaddress = memcpy.toaddress + B;
            memcpy.cpysize = memcpy.cpysize - B;
            memcpy.stagecpysize = memcpy.stagecpysize - B;

UpdateCpyRegisters(memcpy, fault, copied);
```

```
if fault then
    if IsFault(memaddrdesc) then
        AArch64.Abort(memaddrdesc.vaddress, memaddrdesc.fault);

    if IsFault(memstatus) then
        AccessDescriptor accdesc = if iswrite then waccdesc else raccdesc;
        HandleExternalAbort(memstatus, iswrite, memaddrdesc, B, accdesc);

if memcpy.stage == MOPSStage_Prologue then
    PSTATE.<N,Z,C,V> = memcpy.nzcv;
```

C6.2.76 CPYFPRT, CPYFMRT, CPYFERT

Memory Copy Forward-only, reads unprivileged. These instructions perform a memory copy. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYFPRT, then CPYFMRT, and then CPYFERT.

CPYFPRT performs some preconditioning of the arguments suitable for using the CPYFMRT instruction, and performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYFMRT performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYFERT performs the last part of the memory copy.

Note

The inclusion of IMPLEMENTATION DEFINED amounts of memory copy allows some optimization of the size that can be performed.

The memory copy performed by these instructions is in the forward direction only, so the instructions are suitable for a memory copy only where there is no overlap between the source and destination locations, or where the source address is greater than the destination address.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

After execution of CPYFPRT, option A (which results in encoding PSTATE.C = 0):

- If Xn<63> == 1, the copy size is saturated to `0x7FFFFFFFFFFFFFFF`.
- Xs holds the original Xs + saturated Xn.
- Xd holds the original Xd + saturated Xn.
- Xn holds -1* saturated Xn + an IMPLEMENTATION DEFINED number of bytes copied.
- PSTATE.{N,Z,V} are set to {0,0,0}.

After execution of CPYFPRT, option B (which results in encoding PSTATE.C = 1):

- If Xn<63> == 1, the copy size is saturated to `0x7FFFFFFFFFFFFFFF`.
- Xs holds the original Xs + an IMPLEMENTATION DEFINED number of bytes copied.
- Xd holds the original Xd + an IMPLEMENTATION DEFINED number of bytes copied.
- Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
- PSTATE.{N,Z,V} are set to {0,0,0}.

For CPYFMRT, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number and holds -1* the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from -Xn.
- Xd holds the lowest address that the copy is made to -Xn.
- At the end of the instruction, the value of Xn is written back with -1* the number of bytes remaining to be copied in the memory copy in total.

For CPYFMRT, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from.

- Xd holds the lowest address that the copy is copied to.
- At the end of the instruction:
  - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
  - the value of Xs is written back with the lowest address that has not been copied from.
  - the value of Xd is written back with the lowest address that has not been copied to.

For CPYFERT, option A (encoded by PSTATE.C = 0), the format of the arguments is:

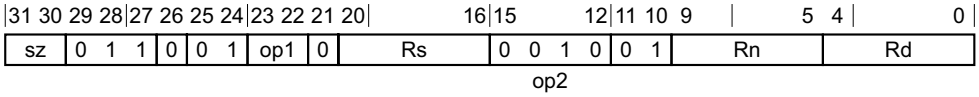
- Xn is treated as a signed 64-bit number and holds -1* the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from -Xn.
- Xd holds the lowest address that the copy is made to -Xn.
- At the end of the instruction, the value of Xn is written back with 0.

For CPYFERT, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from.
- Xd holds the lowest address that the copy is copied to.
- At the end of the instruction:
  - the value of Xn is written back with 0.
  - the value of Xs is written back with the lowest address that has not been copied from.
  - the value of Xd is written back with the lowest address that has not been copied to.

Integer

(FEAT_MOPS)



Epilogue variant

Applies when `op1 == 10`.

```
CPYFERT [<Xd>]!, [<Xs>]!, <Xn>!
```

Main variant

Applies when `op1 == 01`.

```
CPYFMRT [<Xd>]!, [<Xs>]!, <Xn>!
```

Prologue variant

Applies when `op1 == 00`.

```
CPYFPRT [<Xd>]!, [<Xs>]!, <Xn>!
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_MOPS) || sz != '00' then UNDEFINED;

CPYParams memcpy;
memcpy.d = UInt(Rd);
memcpy.s = UInt(Rs);
memcpy.n = UInt(Rn);
bits(4) options = op2;
boolean rnontemporal = options<3> == '1';
boolean wnontemporal = options<2> == '1';

case op1 of
    when '00' memcpy.stage = MOPSStage_Prologue;
    when '01' memcpy.stage = MOPSStage_Main;
    when '10' memcpy.stage = MOPSStage_Epilogue;
    otherwise SEE "Memory Copy and Memory Set";

CheckMOPSEnabled();

if (memcpy.s == memcpy.n || memcpy.s == memcpy.d || memcpy.n == memcpy.d || memcpy.d == 31 || memcpy.s
== 31 || memcpy.n == 31) then
    Constraint c = ConstrainUnpredictable(Unpredictable_MOPSOVERLAP31);
    assert c IN {Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNDEF UNDEFINED;
        when Constraint_NOP   EndOfInstruction();
```

Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*, and particularly *Memory Copy and Memory Set CPY**.

Assembler symbols

<Xd> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the "Rd" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.

<Xs> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the "Rs" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the "Rs" field.

<Xn> For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero at the end of the instruction, encoded in the "Rn" field.

For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the "Rn" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the "Rn" field.

Operation

```
constant integer N = MaxBlockSizeCopiedBytes();
bits(8*N) readdata;

memcpy.nzcv = PSTATE.<N,Z,C,V>;
memcpy.toaddress = X[memcpy.d, 64];
```

```
memcpy.fromaddress = X[memcpy.s, 64];
memcpy.cpysize = SInt(X[memcpy.n, 64]);
memcpy.implements_option_a = CPYFOptionA();

boolean rprivileged = if options<1> == '1' then AArch64.IsUnprivAccessPriv() else PSTATE.EL != EL0;
boolean wprivileged = if options<0> == '1' then AArch64.IsUnprivAccessPriv() else PSTATE.EL != EL0;

AccessDescriptor raccdesc = CreateAccDescMOPS(MemOp_LOAD, rprivileged, rnontemporal);
AccessDescriptor waccdesc = CreateAccDescMOPS(MemOp_STORE, wprivileged, wnontemporal);

if memcpy.stage == MOPSStage_Prologue then
    if memcpy.cpysize<63> == '1' then memcpy.cpysize = 0x7FFFFFFFFFFFFFFF;

    if memcpy.implements_option_a then
        memcpy.nzcv = '0000';
        // Copy in the forward direction offsets the arguments.
        memcpy.toaddress = memcpy.toaddress + memcpy.cpysize;
        memcpy.fromaddress = memcpy.fromaddress + memcpy.cpysize;
        memcpy.cpysize = 0 - memcpy.cpysize;
    else
        memcpy.nzcv = '0010';

memcpy.stagecpysize = MemCpyStageSize(memcpy);

if memcpy.stage != MOPSStage_Prologue then
    CheckMemCpyParams(memcpy, options);

integer copied;
boolean iswrite;
AddressDescriptor memaddrdesc;
PhysMemRetStatus memstatus;
memcpy.forward = TRUE;
boolean fault = FALSE;
integer B;

if memcpy.implements_option_a then
    while memcpy.stagecpysize != 0 && !fault do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(memcpy);
        assert B <= -1 * memcpy.stagecpysize;

        (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress + memcpy.cpysize,
memcpy.fromaddress + memcpy.cpysize, memcpy.forward, B, raccdesc, waccdesc);
        if copied != B then
            fault = TRUE;
        else
            memcpy.cpysize = memcpy.cpysize + B;
            memcpy.stagecpysize = memcpy.stagecpysize + B;
else
    while memcpy.stagecpysize > 0 && !fault do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(memcpy);
        assert B <= memcpy.stagecpysize;

        (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress, memcpy.fromaddress,
memcpy.forward, B, raccdesc, waccdesc);
        if copied != B then
            fault = TRUE;
        else
            memcpy.fromaddress = memcpy.fromaddress + B;
            memcpy.toaddress = memcpy.toaddress + B;
            memcpy.cpysize = memcpy.cpysize - B;
            memcpy.stagecpysize = memcpy.stagecpysize - B;

UpdateCpyRegisters(memcpy, fault, copied);
```

```
if fault then
    if IsFault(memaddrdesc) then
        AArch64.Abort(memaddrdesc.vaddress, memaddrdesc.fault);

    if IsFault(memstatus) then
        AccessDescriptor accdesc = if iswrite then waccdesc else raccdesc;
        HandleExternalAbort(memstatus, iswrite, memaddrdesc, B, accdesc);

if memcpy.stage == MOPSStage_Prologue then
    PSTATE.<N,Z,C,V> = memcpy.nzcv;
```

C6.2.77 CPYFPRTN, CPYFMRTN, CPYFERTN

Memory Copy Forward-only, reads unprivileged, reads and writes non-temporal. These instructions perform a memory copy. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYFPRTN, then CPYFMRTN, and then CPYFERTN.

CPYFPRTN performs some preconditioning of the arguments suitable for using the CPYFMRTN instruction, and performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYFMRTN performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYFERTN performs the last part of the memory copy.

Note

The inclusion of IMPLEMENTATION DEFINED amounts of memory copy allows some optimization of the size that can be performed.

The memory copy performed by these instructions is in the forward direction only, so the instructions are suitable for a memory copy only where there is no overlap between the source and destination locations, or where the source address is greater than the destination address.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

After execution of CPYFPRTN, option A (which results in encoding PSTATE.C = 0):

- If Xn<63> == 1, the copy size is saturated to `0x7FFFFFFFFFFFFFFF`.
- Xs holds the original Xs + saturated Xn.
- Xd holds the original Xd + saturated Xn.
- Xn holds -1* saturated Xn + an IMPLEMENTATION DEFINED number of bytes copied.
- PSTATE.{N,Z,V} are set to {0,0,0}.

After execution of CPYFPRTN, option B (which results in encoding PSTATE.C = 1):

- If Xn<63> == 1, the copy size is saturated to `0x7FFFFFFFFFFFFFFF`.
- Xs holds the original Xs + an IMPLEMENTATION DEFINED number of bytes copied.
- Xd holds the original Xd + an IMPLEMENTATION DEFINED number of bytes copied.
- Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
- PSTATE.{N,Z,V} are set to {0,0,0}.

For CPYFMRTN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number and holds -1* the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from -Xn.
- Xd holds the lowest address that the copy is made to -Xn.
- At the end of the instruction, the value of Xn is written back with -1* the number of bytes remaining to be copied in the memory copy in total.

For CPYFMRTN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from.

- Xd holds the lowest address that the copy is copied to.
- At the end of the instruction:
  - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
  - the value of Xs is written back with the lowest address that has not been copied from.
  - the value of Xd is written back with the lowest address that has not been copied to.

For CPYFERTN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

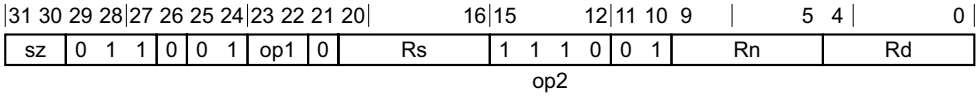
- Xn is treated as a signed 64-bit number and holds -1* the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from -Xn.
- Xd holds the lowest address that the copy is made to -Xn.
- At the end of the instruction, the value of Xn is written back with 0.

For CPYFERTN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from.
- Xd holds the lowest address that the copy is copied to.
- At the end of the instruction:
  - the value of Xn is written back with 0.
  - the value of Xs is written back with the lowest address that has not been copied from.
  - the value of Xd is written back with the lowest address that has not been copied to.

Integer

(FEAT_MOPS)



Epilogue variant

Applies when `op1 == 10`.

```
CPYFERTN [<Xd>]!, [<Xs>]!, <Xn>!
```

Main variant

Applies when `op1 == 01`.

```
CPYFMRTN [<Xd>]!, [<Xs>]!, <Xn>!
```

Prologue variant

Applies when `op1 == 00`.

```
CPYFPRTN [<Xd>]!, [<Xs>]!, <Xn>!
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_MOPS) || sz != '00' then UNDEFINED;

CPYParams memcpy;
memcpy.d = UInt(Rd);
memcpy.s = UInt(Rs);
memcpy.n = UInt(Rn);
bits(4) options = op2;
boolean rnontemporal = options<3> == '1';
boolean wnontemporal = options<2> == '1';

case op1 of
    when '00' memcpy.stage = MOPSStage_Prologue;
    when '01' memcpy.stage = MOPSStage_Main;
    when '10' memcpy.stage = MOPSStage_Epilogue;
    otherwise SEE "Memory Copy and Memory Set";

CheckMOPSEnabled();

if (memcpy.s == memcpy.n || memcpy.s == memcpy.d || memcpy.n == memcpy.d || memcpy.d == 31 || memcpy.s
== 31 || memcpy.n == 31) then
    Constraint c = ConstrainUnpredictable(Unpredictable_MOPSOVERLAP31);
    assert c IN {Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNDEF UNDEFINED;
        when Constraint_NOP   EndOfInstruction();
```

Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*, and particularly *Memory Copy and Memory Set CPY**.

Assembler symbols

<Xd> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the "Rd" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.

<Xs> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the "Rs" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the "Rs" field.

<Xn> For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero at the end of the instruction, encoded in the "Rn" field.

For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the "Rn" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the "Rn" field.

Operation

```
constant integer N = MaxBlockSizeCopiedBytes();
bits(8*N) readdata;

memcpy.nzcv = PSTATE.<N,Z,C,V>;
memcpy.toaddress = X[memcpy.d, 64];
```

```
memcpy.fromaddress = X[memcpy.s, 64];
memcpy.cpysize = SInt(X[memcpy.n, 64]);
memcpy.implements_option_a = CPYFOptionA();

boolean rprivileged = if options<1> == '1' then AArch64.IsUnprivAccessPriv() else PSTATE.EL != EL0;
boolean wprivileged = if options<0> == '1' then AArch64.IsUnprivAccessPriv() else PSTATE.EL != EL0;

AccessDescriptor raccdesc = CreateAccDescMOPS(MemOp_LOAD, rprivileged, rnontemporal);
AccessDescriptor waccdesc = CreateAccDescMOPS(MemOp_STORE, wprivileged, wnontemporal);

if memcpy.stage == MOPSStage_Prologue then
    if memcpy.cpysize<63> == '1' then memcpy.cpysize = 0x7FFFFFFFFFFFFFFF;

    if memcpy.implements_option_a then
        memcpy.nzcv = '0000';
        // Copy in the forward direction offsets the arguments.
        memcpy.toaddress = memcpy.toaddress + memcpy.cpysize;
        memcpy.fromaddress = memcpy.fromaddress + memcpy.cpysize;
        memcpy.cpysize = 0 - memcpy.cpysize;
    else
        memcpy.nzcv = '0010';

memcpy.stagecpysize = MemCpyStageSize(memcpy);

if memcpy.stage != MOPSStage_Prologue then
    CheckMemCpyParams(memcpy, options);

integer copied;
boolean iswrite;
AddressDescriptor memaddrdesc;
PhysMemRetStatus memstatus;
memcpy.forward = TRUE;
boolean fault = FALSE;
integer B;

if memcpy.implements_option_a then
    while memcpy.stagecpysize != 0 && !fault do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(memcpy);
        assert B <= -1 * memcpy.stagecpysize;

        (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress + memcpy.cpysize,
memcpy.fromaddress + memcpy.cpysize, memcpy.forward, B, raccdesc, waccdesc);
        if copied != B then
            fault = TRUE;
        else
            memcpy.cpysize = memcpy.cpysize + B;
            memcpy.stagecpysize = memcpy.stagecpysize + B;
else
    while memcpy.stagecpysize > 0 && !fault do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(memcpy);
        assert B <= memcpy.stagecpysize;

        (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress, memcpy.fromaddress,
memcpy.forward, B, raccdesc, waccdesc);
        if copied != B then
            fault = TRUE;
        else
            memcpy.fromaddress = memcpy.fromaddress + B;
            memcpy.toaddress = memcpy.toaddress + B;
            memcpy.cpysize = memcpy.cpysize - B;
            memcpy.stagecpysize = memcpy.stagecpysize - B;

UpdateCpyRegisters(memcpy, fault, copied);
```

```
if fault then
    if IsFault(memaddrdesc) then
        AArch64.Abort(memaddrdesc.vaddress, memaddrdesc.fault);

    if IsFault(memstatus) then
        AccessDescriptor accdesc = if iswrite then waccdesc else raccdesc;
        HandleExternalAbort(memstatus, iswrite, memaddrdesc, B, accdesc);

if memcpy.stage == MOPSStage_Prologue then
    PSTATE.<N,Z,C,V> = memcpy.nzcv;
```

C6.2.78 CPYFPRTRN, CPYFMRTRN, CPYFERTRN

Memory Copy Forward-only, reads unprivileged and non-temporal. These instructions perform a memory copy. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYFPRTRN, then CPYFMRTRN, and then CPYFERTRN.

CPYFPRTRN performs some preconditioning of the arguments suitable for using the CPYFMRTRN instruction, and performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYFMRTRN performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYFERTRN performs the last part of the memory copy.

Note

The inclusion of IMPLEMENTATION DEFINED amounts of memory copy allows some optimization of the size that can be performed.

The memory copy performed by these instructions is in the forward direction only, so the instructions are suitable for a memory copy only where there is no overlap between the source and destination locations, or where the source address is greater than the destination address.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

After execution of CPYFPRTRN, option A (which results in encoding PSTATE.C = 0):

- If Xn<63> == 1, the copy size is saturated to `0x7FFFFFFFFFFFFFFF`.
- Xs holds the original Xs + saturated Xn.
- Xd holds the original Xd + saturated Xn.
- Xn holds -1* saturated Xn + an IMPLEMENTATION DEFINED number of bytes copied.
- PSTATE.{N,Z,V} are set to {0,0,0}.

After execution of CPYFPRTRN, option B (which results in encoding PSTATE.C = 1):

- If Xn<63> == 1, the copy size is saturated to `0x7FFFFFFFFFFFFFFF`.
- Xs holds the original Xs + an IMPLEMENTATION DEFINED number of bytes copied.
- Xd holds the original Xd + an IMPLEMENTATION DEFINED number of bytes copied.
- Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
- PSTATE.{N,Z,V} are set to {0,0,0}.

For CPYFMRTRN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number and holds -1* the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from -Xn.
- Xd holds the lowest address that the copy is made to -Xn.
- At the end of the instruction, the value of Xn is written back with -1* the number of bytes remaining to be copied in the memory copy in total.

For CPYFMRTRN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from.

- Xd holds the lowest address that the copy is copied to.
- At the end of the instruction:
  - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
  - the value of Xs is written back with the lowest address that has not been copied from.
  - the value of Xd is written back with the lowest address that has not been copied to.

For CPYFERTRN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

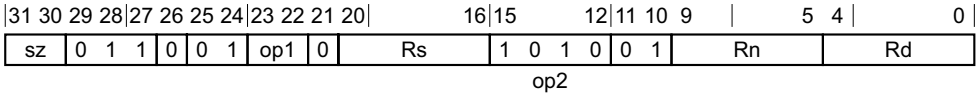
- Xn is treated as a signed 64-bit number and holds -1* the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from -Xn.
- Xd holds the lowest address that the copy is made to -Xn.
- At the end of the instruction, the value of Xn is written back with 0.

For CPYFERTRN option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from.
- Xd holds the lowest address that the copy is copied to.
- At the end of the instruction:
  - the value of Xn is written back with 0.
  - the value of Xs is written back with the lowest address that has not been copied from.
  - the value of Xd is written back with the lowest address that has not been copied to.

Integer

(FEAT_MOPS)



Epilogue variant

Applies when `op1 == 10`.

```
CPYFERTRN [<Xd>]!, [<Xs>]!, <Xn>!
```

Main variant

Applies when `op1 == 01`.

```
CPYFMRTRN [<Xd>]!, [<Xs>]!, <Xn>!
```

Prologue variant

Applies when `op1 == 00`.

```
CPYFPRTRN [<Xd>]!, [<Xs>]!, <Xn>!
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_MOPS) || sz != '00' then UNDEFINED;

CPYParams memcpy;
memcpy.d = UInt(Rd);
memcpy.s = UInt(Rs);
memcpy.n = UInt(Rn);
bits(4) options = op2;
boolean rnontemporal = options<3> == '1';
boolean wnontemporal = options<2> == '1';

case op1 of
    when '00' memcpy.stage = MOPSStage_Prologue;
    when '01' memcpy.stage = MOPSStage_Main;
    when '10' memcpy.stage = MOPSStage_Epilogue;
    otherwise SEE "Memory Copy and Memory Set";

CheckMOPSEnabled();

if (memcpy.s == memcpy.n || memcpy.s == memcpy.d || memcpy.n == memcpy.d || memcpy.d == 31 || memcpy.s
== 31 || memcpy.n == 31) then
    Constraint c = ConstrainUnpredictable(Unpredictable_MOPSOVERLAP31);
    assert c IN {Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNDEF UNDEFINED;
        when Constraint_NOP   EndOfInstruction();
```

Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*, and particularly *Memory Copy and Memory Set CPY**.

Assembler symbols

<Xd> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the "Rd" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.

<Xs> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the "Rs" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the "Rs" field.

<Xn> For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero at the end of the instruction, encoded in the "Rn" field.

For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the "Rn" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the "Rn" field.

Operation

```
constant integer N = MaxBlockSizeCopiedBytes();
bits(8*N) readdata;

memcpy.nzcv = PSTATE.<N,Z,C,V>;
memcpy.toaddress = X[memcpy.d, 64];
```

```
memcpy.fromaddress = X[memcpy.s, 64];
memcpy.cpysize = SInt(X[memcpy.n, 64]);
memcpy.implements_option_a = CPYFOptionA();

boolean rprivileged = if options<1> == '1' then AArch64.IsUnprivAccessPriv() else PSTATE.EL != EL0;
boolean wprivileged = if options<0> == '1' then AArch64.IsUnprivAccessPriv() else PSTATE.EL != EL0;

AccessDescriptor raccdesc = CreateAccDescMOPS(MemOp_LOAD, rprivileged, rnontemporal);
AccessDescriptor waccdesc = CreateAccDescMOPS(MemOp_STORE, wprivileged, wnontemporal);

if memcpy.stage == MOPSStage_Prologue then
    if memcpy.cpysize<63> == '1' then memcpy.cpysize = 0x7FFFFFFFFFFFFFFF;

    if memcpy.implements_option_a then
        memcpy.nzcv = '0000';
        // Copy in the forward direction offsets the arguments.
        memcpy.toaddress = memcpy.toaddress + memcpy.cpysize;
        memcpy.fromaddress = memcpy.fromaddress + memcpy.cpysize;
        memcpy.cpysize = 0 - memcpy.cpysize;
    else
        memcpy.nzcv = '0010';

memcpy.stagecpysize = MemCpyStageSize(memcpy);

if memcpy.stage != MOPSStage_Prologue then
    CheckMemCpyParams(memcpy, options);

integer copied;
boolean iswrite;
AddressDescriptor memaddrdesc;
PhysMemRetStatus memstatus;
memcpy.forward = TRUE;
boolean fault = FALSE;
integer B;

if memcpy.implements_option_a then
    while memcpy.stagecpysize != 0 && !fault do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(memcpy);
        assert B <= -1 * memcpy.stagecpysize;

        (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress + memcpy.cpysize,
memcpy.fromaddress + memcpy.cpysize, memcpy.forward, B, raccdesc, waccdesc);
        if copied != B then
            fault = TRUE;
        else
            memcpy.cpysize = memcpy.cpysize + B;
            memcpy.stagecpysize = memcpy.stagecpysize + B;
else
    while memcpy.stagecpysize > 0 && !fault do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(memcpy);
        assert B <= memcpy.stagecpysize;

        (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress, memcpy.fromaddress,
memcpy.forward, B, raccdesc, waccdesc);
        if copied != B then
            fault = TRUE;
        else
            memcpy.fromaddress = memcpy.fromaddress + B;
            memcpy.toaddress = memcpy.toaddress + B;
            memcpy.cpysize = memcpy.cpysize - B;
            memcpy.stagecpysize = memcpy.stagecpysize - B;

UpdateCpyRegisters(memcpy, fault, copied);
```

```
if fault then
    if IsFault(memaddrdesc) then
        AArch64.Abort(memaddrdesc.vaddress, memaddrdesc.fault);

    if IsFault(memstatus) then
        AccessDescriptor accdesc = if iswrite then waccdesc else raccdesc;
        HandleExternalAbort(memstatus, iswrite, memaddrdesc, B, accdesc);

if memcpy.stage == MOPSStage_Prologue then
    PSTATE.<N,Z,C,V> = memcpy.nzcv;
```

C6.2.79 CPYFPRTWN, CPYFMRTWN, CPYFERTWN

Memory Copy Forward-only, reads unprivileged, writes non-temporal. These instructions perform a memory copy. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYFPRTWN, then CPYFMRTWN, and then CPYFERTWN.

CPYFPRTWN performs some preconditioning of the arguments suitable for using the CPYFMRTWN instruction, and performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYFMRTWN performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYFERTWN performs the last part of the memory copy.

Note

The inclusion of IMPLEMENTATION DEFINED amounts of memory copy allows some optimization of the size that can be performed.

The memory copy performed by these instructions is in the forward direction only, so the instructions are suitable for a memory copy only where there is no overlap between the source and destination locations, or where the source address is greater than the destination address.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

After execution of CPYFPRTWN, option A (which results in encoding PSTATE.C = 0):

- If Xn<63> == 1, the copy size is saturated to `0x7FFFFFFFFFFFFFFF`.
- Xs holds the original Xs + saturated Xn.
- Xd holds the original Xd + saturated Xn.
- Xn holds -1* saturated Xn + an IMPLEMENTATION DEFINED number of bytes copied.
- PSTATE.{N,Z,V} are set to {0,0,0}.

After execution of CPYFPRTWN, option B (which results in encoding PSTATE.C = 1):

- If Xn<63> == 1, the copy size is saturated to `0x7FFFFFFFFFFFFFFF`.
- Xs holds the original Xs + an IMPLEMENTATION DEFINED number of bytes copied.
- Xd holds the original Xd + an IMPLEMENTATION DEFINED number of bytes copied.
- Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
- PSTATE.{N,Z,V} are set to {0,0,0}.

For CPYFMRTWN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number and holds -1* the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from -Xn.
- Xd holds the lowest address that the copy is made to -Xn.
- At the end of the instruction, the value of Xn is written back with -1* the number of bytes remaining to be copied in the memory copy in total.

For CPYFMRTWN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from.

- Xd holds the lowest address that the copy is copied to.
- At the end of the instruction:
  - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
  - the value of Xs is written back with the lowest address that has not been copied from.
  - the value of Xd is written back with the lowest address that has not been copied to.

For CPYFERTWN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

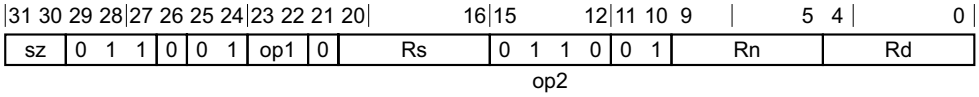
- Xn is treated as a signed 64-bit number and holds -1* the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from -Xn.
- Xd holds the lowest address that the copy is made to -Xn.
- At the end of the instruction, the value of Xn is written back with 0.

For CPYFERTWN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from.
- Xd holds the lowest address that the copy is copied to.
- At the end of the instruction:
  - the value of Xn is written back with 0.
  - the value of Xs is written back with the lowest address that has not been copied from.
  - the value of Xd is written back with the lowest address that has not been copied to.

Integer

(FEAT_MOPS)



Epilogue variant

Applies when `op1 == 10`.

```
CPYFERTWN [<Xd>]!, [<Xs>]!, <Xn>!
```

Main variant

Applies when `op1 == 01`.

```
CPYFMRTWN [<Xd>]!, [<Xs>]!, <Xn>!
```

Prologue variant

Applies when `op1 == 00`.

```
CPYFPRTWN [<Xd>]!, [<Xs>]!, <Xn>!
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_MOPS) || sz != '00' then UNDEFINED;

CPYParams memcpy;
memcpy.d = UInt(Rd);
memcpy.s = UInt(Rs);
memcpy.n = UInt(Rn);
bits(4) options = op2;
boolean rnontemporal = options<3> == '1';
boolean wnontemporal = options<2> == '1';

case op1 of
    when '00' memcpy.stage = MOPSStage_Prologue;
    when '01' memcpy.stage = MOPSStage_Main;
    when '10' memcpy.stage = MOPSStage_Epilogue;
    otherwise SEE "Memory Copy and Memory Set";

CheckMOPSEnabled();

if (memcpy.s == memcpy.n || memcpy.s == memcpy.d || memcpy.n == memcpy.d || memcpy.d == 31 || memcpy.s
== 31 || memcpy.n == 31) then
    Constraint c = ConstrainUnpredictable(Unpredictable_MOPSOVERLAP31);
    assert c IN {Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNDEF UNDEFINED;
        when Constraint_NOP   EndOfInstruction();
```

Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*, and particularly *Memory Copy and Memory Set CPY**.

Assembler symbols

<Xd> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the "Rd" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.

<Xs> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the "Rs" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the "Rs" field.

<Xn> For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero at the end of the instruction, encoded in the "Rn" field.

For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the "Rn" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the "Rn" field.

Operation

```
constant integer N = MaxBlockSizeCopiedBytes();
bits(8*N) readdata;

memcpy.nzcv = PSTATE.<N,Z,C,V>;
memcpy.toaddress = X[memcpy.d, 64];
```

```
memcpy.fromaddress = X[memcpy.s, 64];
memcpy.cpysize = SInt(X[memcpy.n, 64]);
memcpy.implements_option_a = CPYFOptionA();

boolean rprivileged = if options<1> == '1' then AArch64.IsUnprivAccessPriv() else PSTATE.EL != EL0;
boolean wprivileged = if options<0> == '1' then AArch64.IsUnprivAccessPriv() else PSTATE.EL != EL0;

AccessDescriptor raccdesc = CreateAccDescMOPS(MemOp_LOAD, rprivileged, rnontemporal);
AccessDescriptor waccdesc = CreateAccDescMOPS(MemOp_STORE, wprivileged, wnontemporal);

if memcpy.stage == MOPSStage_Prologue then
    if memcpy.cpysize<63> == '1' then memcpy.cpysize = 0x7FFFFFFFFFFFFFFF;

    if memcpy.implements_option_a then
        memcpy.nzcv = '0000';
        // Copy in the forward direction offsets the arguments.
        memcpy.toaddress = memcpy.toaddress + memcpy.cpysize;
        memcpy.fromaddress = memcpy.fromaddress + memcpy.cpysize;
        memcpy.cpysize = 0 - memcpy.cpysize;
    else
        memcpy.nzcv = '0010';

memcpy.stagecpysize = MemCpyStageSize(memcpy);

if memcpy.stage != MOPSStage_Prologue then
    CheckMemCpyParams(memcpy, options);

integer copied;
boolean iswrite;
AddressDescriptor memaddrdesc;
PhysMemRetStatus memstatus;
memcpy.forward = TRUE;
boolean fault = FALSE;
integer B;

if memcpy.implements_option_a then
    while memcpy.stagecpysize != 0 && !fault do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(memcpy);
        assert B <= -1 * memcpy.stagecpysize;

        (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress + memcpy.cpysize,
memcpy.fromaddress + memcpy.cpysize, memcpy.forward, B, raccdesc, waccdesc);
        if copied != B then
            fault = TRUE;
        else
            memcpy.cpysize = memcpy.cpysize + B;
            memcpy.stagecpysize = memcpy.stagecpysize + B;
else
    while memcpy.stagecpysize > 0 && !fault do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(memcpy);
        assert B <= memcpy.stagecpysize;

        (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress, memcpy.fromaddress,
memcpy.forward, B, raccdesc, waccdesc);
        if copied != B then
            fault = TRUE;
        else
            memcpy.fromaddress = memcpy.fromaddress + B;
            memcpy.toaddress = memcpy.toaddress + B;
            memcpy.cpysize = memcpy.cpysize - B;
            memcpy.stagecpysize = memcpy.stagecpysize - B;

UpdateCpyRegisters(memcpy, fault, copied);
```

```
if fault then
    if IsFault(memaddrdesc) then
        AArch64.Abort(memaddrdesc.vaddress, memaddrdesc.fault);

    if IsFault(memstatus) then
        AccessDescriptor accdesc = if iswrite then waccdesc else raccdesc;
        HandleExternalAbort(memstatus, iswrite, memaddrdesc, B, accdesc);

if memcpy.stage == MOPSStage_Prologue then
    PSTATE.<N,Z,C,V> = memcpy.nzcv;
```

C6.2.80 CPYFPT, CPYFMT, CPYFET

Memory Copy Forward-only, reads and writes unprivileged. These instructions perform a memory copy. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYFPT, then CPYFMT, and then CPYFET.

CPYFPT performs some preconditioning of the arguments suitable for using the CPYFMT instruction, and performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYFMT performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYFET performs the last part of the memory copy.

Note

The inclusion of IMPLEMENTATION DEFINED amounts of memory copy allows some optimization of the size that can be performed.

The memory copy performed by these instructions is in the forward direction only, so the instructions are suitable for a memory copy only where there is no overlap between the source and destination locations, or where the source address is greater than the destination address.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

After execution of CPYFPT, option A (which results in encoding PSTATE.C = 0):

- If Xn<63> == 1, the copy size is saturated to `0x7FFFFFFFFFFFFFFF`.
- Xs holds the original Xs + saturated Xn.
- Xd holds the original Xd + saturated Xn.
- Xn holds -1* saturated Xn + an IMPLEMENTATION DEFINED number of bytes copied.
- PSTATE.{N,Z,V} are set to {0,0,0}.

After execution of CPYFPT, option B (which results in encoding PSTATE.C = 1):

- If Xn<63> == 1, the copy size is saturated to `0x7FFFFFFFFFFFFFFF`.
- Xs holds the original Xs + an IMPLEMENTATION DEFINED number of bytes copied.
- Xd holds the original Xd + an IMPLEMENTATION DEFINED number of bytes copied.
- Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
- PSTATE.{N,Z,V} are set to {0,0,0}.

For CPYFMT, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number and holds -1* the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from -Xn.
- Xd holds the lowest address that the copy is made to -Xn.
- At the end of the instruction, the value of Xn is written back with -1* the number of bytes remaining to be copied in the memory copy in total.

For CPYFMT, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from.

- Xd holds the lowest address that the copy is copied to.
- At the end of the instruction:
  - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
  - the value of Xs is written back with the lowest address that has not been copied from.
  - the value of Xd is written back with the lowest address that has not been copied to.

For CPYFET, option A (encoded by PSTATE.C = 0), the format of the arguments is:

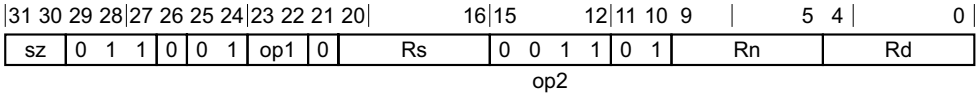
- Xn is treated as a signed 64-bit number and holds -1* the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from -Xn.
- Xd holds the lowest address that the copy is made to -Xn.
- At the end of the instruction, the value of Xn is written back with 0.

For CPYFET, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from.
- Xd holds the lowest address that the copy is copied to.
- At the end of the instruction:
  - the value of Xn is written back with 0.
  - the value of Xs is written back with the lowest address that has not been copied from.
  - the value of Xd is written back with the lowest address that has not been copied to.

Integer

(FEAT_MOPS)



Epilogue variant

Applies when `op1 == 10`.

```
CPYFET [<Xd>]!, [<Xs>]!, <Xn>!
```

Main variant

Applies when `op1 == 01`.

```
CPYFMT [<Xd>]!, [<Xs>]!, <Xn>!
```

Prologue variant

Applies when `op1 == 00`.

```
CPYFPT [<Xd>]!, [<Xs>]!, <Xn>!
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_MOPS) || sz != '00' then UNDEFINED;

CPYParams memcpy;
memcpy.d = UInt(Rd);
memcpy.s = UInt(Rs);
memcpy.n = UInt(Rn);
bits(4) options = op2;
boolean rnontemporal = options<3> == '1';
boolean wnontemporal = options<2> == '1';

case op1 of
    when '00' memcpy.stage = MOPSStage_Prologue;
    when '01' memcpy.stage = MOPSStage_Main;
    when '10' memcpy.stage = MOPSStage_Epilogue;
    otherwise SEE "Memory Copy and Memory Set";

CheckMOPSEnabled();

if (memcpy.s == memcpy.n || memcpy.s == memcpy.d || memcpy.n == memcpy.d || memcpy.d == 31 || memcpy.s
== 31 || memcpy.n == 31) then
    Constraint c = ConstrainUnpredictable(Unpredictable_MOPSOVERLAP31);
    assert c IN {Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNDEF UNDEFINED;
        when Constraint_NOP   EndOfInstruction();
```

Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*, and particularly *Memory Copy and Memory Set CPY**.

Assembler symbols

<Xd> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the "Rd" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.

<Xs> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the "Rs" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the "Rs" field.

<Xn> For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero at the end of the instruction, encoded in the "Rn" field.

For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the "Rn" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the "Rn" field.

Operation

```
constant integer N = MaxBlockSizeCopiedBytes();
bits(8*N) readdata;

memcpy.nzcv = PSTATE.<N,Z,C,V>;
memcpy.toaddress = X[memcpy.d, 64];
```

```
memcpy.fromaddress = X[memcpy.s, 64];
memcpy.cpysize = SInt(X[memcpy.n, 64]);
memcpy.implements_option_a = CPYFOptionA();

boolean rprivileged = if options<1> == '1' then AArch64.IsUnprivAccessPriv() else PSTATE.EL != EL0;
boolean wprivileged = if options<0> == '1' then AArch64.IsUnprivAccessPriv() else PSTATE.EL != EL0;

AccessDescriptor raccdesc = CreateAccDescMOPS(MemOp_LOAD, rprivileged, rnontemporal);
AccessDescriptor waccdesc = CreateAccDescMOPS(MemOp_STORE, wprivileged, wnontemporal);

if memcpy.stage == MOPSStage_Prologue then
    if memcpy.cpysize<63> == '1' then memcpy.cpysize = 0x7FFFFFFFFFFFFFFF;

    if memcpy.implements_option_a then
        memcpy.nzcv = '0000';
        // Copy in the forward direction offsets the arguments.
        memcpy.toaddress = memcpy.toaddress + memcpy.cpysize;
        memcpy.fromaddress = memcpy.fromaddress + memcpy.cpysize;
        memcpy.cpysize = 0 - memcpy.cpysize;
    else
        memcpy.nzcv = '0010';

memcpy.stagecpysize = MemCpyStageSize(memcpy);

if memcpy.stage != MOPSStage_Prologue then
    CheckMemCpyParams(memcpy, options);

integer copied;
boolean iswrite;
AddressDescriptor memaddrdesc;
PhysMemRetStatus memstatus;
memcpy.forward = TRUE;
boolean fault = FALSE;
integer B;

if memcpy.implements_option_a then
    while memcpy.stagecpysize != 0 && !fault do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(memcpy);
        assert B <= -1 * memcpy.stagecpysize;

        (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress + memcpy.cpysize,
memcpy.fromaddress + memcpy.cpysize, memcpy.forward, B, raccdesc, waccdesc);
        if copied != B then
            fault = TRUE;
        else
            memcpy.cpysize = memcpy.cpysize + B;
            memcpy.stagecpysize = memcpy.stagecpysize + B;
else
    while memcpy.stagecpysize > 0 && !fault do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(memcpy);
        assert B <= memcpy.stagecpysize;

        (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress, memcpy.fromaddress,
memcpy.forward, B, raccdesc, waccdesc);
        if copied != B then
            fault = TRUE;
        else
            memcpy.fromaddress = memcpy.fromaddress + B;
            memcpy.toaddress = memcpy.toaddress + B;
            memcpy.cpysize = memcpy.cpysize - B;
            memcpy.stagecpysize = memcpy.stagecpysize - B;

UpdateCpyRegisters(memcpy, fault, copied);
```

```
if fault then
    if IsFault(memaddrdesc) then
        AArch64.Abort(memaddrdesc.vaddress, memaddrdesc.fault);

    if IsFault(memstatus) then
        AccessDescriptor accdesc = if iswrite then waccdesc else raccdesc;
        HandleExternalAbort(memstatus, iswrite, memaddrdesc, B, accdesc);

if memcpy.stage == MOPSStage_Prologue then
    PSTATE.<N,Z,C,V> = memcpy.nzcv;
```

C6.2.81 CPYFPTN, CPYFMTN, CPYFETN

Memory Copy Forward-only, reads and writes unprivileged and non-temporal. These instructions perform a memory copy. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYFPTN, then CPYFMTN, and then CPYFETN.

CPYFPTN performs some preconditioning of the arguments suitable for using the CPYFMTN instruction, and performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYFMTN performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYFETN performs the last part of the memory copy.

Note

The inclusion of IMPLEMENTATION DEFINED amounts of memory copy allows some optimization of the size that can be performed.

The memory copy performed by these instructions is in the forward direction only, so the instructions are suitable for a memory copy only where there is no overlap between the source and destination locations, or where the source address is greater than the destination address.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

After execution of CPYFPTN, option A (which results in encoding PSTATE.C = 0):

- If Xn<63> == 1, the copy size is saturated to `0x7FFFFFFFFFFFFFFF`.
- Xs holds the original Xs + saturated Xn.
- Xd holds the original Xd + saturated Xn.
- Xn holds -1* saturated Xn + an IMPLEMENTATION DEFINED number of bytes copied.
- PSTATE.{N,Z,V} are set to {0,0,0}.

After execution of CPYFPTN, option B (which results in encoding PSTATE.C = 1):

- If Xn<63> == 1, the copy size is saturated to `0x7FFFFFFFFFFFFFFF`.
- Xs holds the original Xs + an IMPLEMENTATION DEFINED number of bytes copied.
- Xd holds the original Xd + an IMPLEMENTATION DEFINED number of bytes copied.
- Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
- PSTATE.{N,Z,V} are set to {0,0,0}.

For CPYFMTN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number and holds -1* the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from -Xn.
- Xd holds the lowest address that the copy is made to -Xn.
- At the end of the instruction, the value of Xn is written back with -1* the number of bytes remaining to be copied in the memory copy in total.

For CPYFMTN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from.

- Xd holds the lowest address that the copy is copied to.
- At the end of the instruction:
  - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
  - the value of Xs is written back with the lowest address that has not been copied from.
  - the value of Xd is written back with the lowest address that has not been copied to.

For CPYFETN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

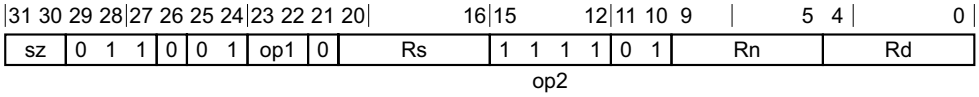
- Xn is treated as a signed 64-bit number and holds -1* the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from -Xn.
- Xd holds the lowest address that the copy is made to -Xn.
- At the end of the instruction, the value of Xn is written back with 0.

For CPYFETN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from.
- Xd holds the lowest address that the copy is copied to.
- At the end of the instruction:
  - the value of Xn is written back with 0.
  - the value of Xs is written back with the lowest address that has not been copied from.
  - the value of Xd is written back with the lowest address that has not been copied to.

Integer

(FEAT_MOPS)



Epilogue variant

Applies when `op1 == 10`.

```
CPYFETN [<Xd>]!, [<Xs>]!, <Xn>!
```

Main variant

Applies when `op1 == 01`.

```
CPYFMTN [<Xd>]!, [<Xs>]!, <Xn>!
```

Prologue variant

Applies when `op1 == 00`.

```
CPYFPTN [<Xd>]!, [<Xs>]!, <Xn>!
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_MOPS) || sz != '00' then UNDEFINED;

CPYParams memcpy;
memcpy.d = UInt(Rd);
memcpy.s = UInt(Rs);
memcpy.n = UInt(Rn);
bits(4) options = op2;
boolean rnontemporal = options<3> == '1';
boolean wnontemporal = options<2> == '1';

case op1 of
    when '00' memcpy.stage = MOPSStage_Prologue;
    when '01' memcpy.stage = MOPSStage_Main;
    when '10' memcpy.stage = MOPSStage_Epilogue;
    otherwise SEE "Memory Copy and Memory Set";

CheckMOPSEnabled();

if (memcpy.s == memcpy.n || memcpy.s == memcpy.d || memcpy.n == memcpy.d || memcpy.d == 31 || memcpy.s
== 31 || memcpy.n == 31) then
    Constraint c = ConstrainUnpredictable(Unpredictable_MOPSOVERLAP31);
    assert c IN {Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNDEF UNDEFINED;
        when Constraint_NOP   EndOfInstruction();
```

Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*, and particularly *Memory Copy and Memory Set CPY**.

Assembler symbols

<Xd> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the "Rd" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.

<Xs> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the "Rs" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the "Rs" field.

<Xn> For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero at the end of the instruction, encoded in the "Rn" field.

For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the "Rn" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the "Rn" field.

Operation

```
constant integer N = MaxBlockSizeCopiedBytes();
bits(8*N) readdata;

memcpy.nzcv = PSTATE.<N,Z,C,V>;
memcpy.toaddress = X[memcpy.d, 64];
```

```
memcpy.fromaddress = X[memcpy.s, 64];
memcpy.cpysize = SInt(X[memcpy.n, 64]);
memcpy.implements_option_a = CPYFOptionA();

boolean rprivileged = if options<1> == '1' then AArch64.IsUnprivAccessPriv() else PSTATE.EL != EL0;
boolean wprivileged = if options<0> == '1' then AArch64.IsUnprivAccessPriv() else PSTATE.EL != EL0;

AccessDescriptor raccdesc = CreateAccDescMOPS(MemOp_LOAD, rprivileged, rnontemporal);
AccessDescriptor waccdesc = CreateAccDescMOPS(MemOp_STORE, wprivileged, wnontemporal);

if memcpy.stage == MOPSStage_Prologue then
    if memcpy.cpysize<63> == '1' then memcpy.cpysize = 0x7FFFFFFFFFFFFFFF;

    if memcpy.implements_option_a then
        memcpy.nzcv = '0000';
        // Copy in the forward direction offsets the arguments.
        memcpy.toaddress = memcpy.toaddress + memcpy.cpysize;
        memcpy.fromaddress = memcpy.fromaddress + memcpy.cpysize;
        memcpy.cpysize = 0 - memcpy.cpysize;
    else
        memcpy.nzcv = '0010';

memcpy.stagecpysize = MemCpyStageSize(memcpy);

if memcpy.stage != MOPSStage_Prologue then
    CheckMemCpyParams(memcpy, options);

integer copied;
boolean iswrite;
AddressDescriptor memaddrdesc;
PhysMemRetStatus memstatus;
memcpy.forward = TRUE;
boolean fault = FALSE;
integer B;

if memcpy.implements_option_a then
    while memcpy.stagecpysize != 0 && !fault do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(memcpy);
        assert B <= -1 * memcpy.stagecpysize;

        (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress + memcpy.cpysize,
memcpy.fromaddress + memcpy.cpysize, memcpy.forward, B, raccdesc, waccdesc);
        if copied != B then
            fault = TRUE;
        else
            memcpy.cpysize = memcpy.cpysize + B;
            memcpy.stagecpysize = memcpy.stagecpysize + B;
else
    while memcpy.stagecpysize > 0 && !fault do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(memcpy);
        assert B <= memcpy.stagecpysize;

        (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress, memcpy.fromaddress,
memcpy.forward, B, raccdesc, waccdesc);
        if copied != B then
            fault = TRUE;
        else
            memcpy.fromaddress = memcpy.fromaddress + B;
            memcpy.toaddress = memcpy.toaddress + B;
            memcpy.cpysize = memcpy.cpysize - B;
            memcpy.stagecpysize = memcpy.stagecpysize - B;

UpdateCpyRegisters(memcpy, fault, copied);
```

```
if fault then
    if IsFault(memaddrdesc) then
        AArch64.Abort(memaddrdesc.vaddress, memaddrdesc.fault);

    if IsFault(memstatus) then
        AccessDescriptor accdesc = if iswrite then waccdesc else raccdesc;
        HandleExternalAbort(memstatus, iswrite, memaddrdesc, B, accdesc);

if memcpy.stage == MOPSStage_Prologue then
    PSTATE.<N,Z,C,V> = memcpy.nzcv;
```

C6.2.82 CPYFPTRN, CPYFMTRN, CPYFETRN

Memory Copy Forward-only, reads and writes unprivileged, reads non-temporal. These instructions perform a memory copy. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYFPTRN, then CPYFMTRN, and then CPYFETRN.

CPYFPTRN performs some preconditioning of the arguments suitable for using the CPYFMTRN instruction, and performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYFMTRN performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYFETRN performs the last part of the memory copy.

Note

The inclusion of IMPLEMENTATION DEFINED amounts of memory copy allows some optimization of the size that can be performed.

The memory copy performed by these instructions is in the forward direction only, so the instructions are suitable for a memory copy only where there is no overlap between the source and destination locations, or where the source address is greater than the destination address.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

After execution of CPYFPTRN, option A (which results in encoding PSTATE.C = 0):

- If Xn<63> == 1, the copy size is saturated to `0x7FFFFFFFFFFFFFFF`.
- Xs holds the original Xs + saturated Xn.
- Xd holds the original Xd + saturated Xn.
- Xn holds -1* saturated Xn + an IMPLEMENTATION DEFINED number of bytes copied.
- PSTATE.{N,Z,V} are set to {0,0,0}.

After execution of CPYFPTRN, option B (which results in encoding PSTATE.C = 1):

- If Xn<63> == 1, the copy size is saturated to `0x7FFFFFFFFFFFFFFF`.
- Xs holds the original Xs + an IMPLEMENTATION DEFINED number of bytes copied.
- Xd holds the original Xd + an IMPLEMENTATION DEFINED number of bytes copied.
- Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
- PSTATE.{N,Z,V} are set to {0,0,0}.

For CPYFMTRN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number and holds -1* the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from -Xn.
- Xd holds the lowest address that the copy is made to -Xn.
- At the end of the instruction, the value of Xn is written back with -1* the number of bytes remaining to be copied in the memory copy in total.

For CPYFMTRN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from.

- Xd holds the lowest address that the copy is copied to.
- At the end of the instruction:
  - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
  - the value of Xs is written back with the lowest address that has not been copied from.
  - the value of Xd is written back with the lowest address that has not been copied to.

For CPYFETRN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

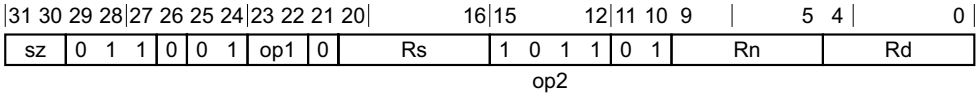
- Xn is treated as a signed 64-bit number and holds -1* the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from -Xn.
- Xd holds the lowest address that the copy is made to -Xn.
- At the end of the instruction, the value of Xn is written back with 0.

For CPYFETRN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from.
- Xd holds the lowest address that the copy is copied to.
- At the end of the instruction:
  - the value of Xn is written back with 0.
  - the value of Xs is written back with the lowest address that has not been copied from.
  - the value of Xd is written back with the lowest address that has not been copied to.

Integer

(FEAT_MOPS)



Epilogue variant

Applies when `op1 == 10`.

```
CPYFETRN [<Xd>]!, [<Xs>]!, <Xn>!
```

Main variant

Applies when `op1 == 01`.

```
CPYFMTRN [<Xd>]!, [<Xs>]!, <Xn>!
```

Prologue variant

Applies when `op1 == 00`.

```
CPYFPTRN [<Xd>]!, [<Xs>]!, <Xn>!
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_MOPS) || sz != '00' then UNDEFINED;

CPYParams memcpy;
memcpy.d = UInt(Rd);
memcpy.s = UInt(Rs);
memcpy.n = UInt(Rn);
bits(4) options = op2;
boolean rnontemporal = options<3> == '1';
boolean wnontemporal = options<2> == '1';

case op1 of
    when '00' memcpy.stage = MOPSStage_Prologue;
    when '01' memcpy.stage = MOPSStage_Main;
    when '10' memcpy.stage = MOPSStage_Epilogue;
    otherwise SEE "Memory Copy and Memory Set";

CheckMOPSEnabled();

if (memcpy.s == memcpy.n || memcpy.s == memcpy.d || memcpy.n == memcpy.d || memcpy.d == 31 || memcpy.s
== 31 || memcpy.n == 31) then
    Constraint c = ConstrainUnpredictable(Unpredictable_MOPSOVERLAP31);
    assert c IN {Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNDEF UNDEFINED;
        when Constraint_NOP   EndOfInstruction();
```

Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*, and particularly *Memory Copy and Memory Set CPY**.

Assembler symbols

<Xd> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the "Rd" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.

<Xs> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the "Rs" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the "Rs" field.

<Xn> For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero at the end of the instruction, encoded in the "Rn" field.

For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the "Rn" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the "Rn" field.

Operation

```
constant integer N = MaxBlockSizeCopiedBytes();
bits(8*N) readdata;

memcpy.nzcv = PSTATE.<N,Z,C,V>;
memcpy.toaddress = X[memcpy.d, 64];
```

```
memcpy.fromaddress = X[memcpy.s, 64];
memcpy.cpysize = SInt(X[memcpy.n, 64]);
memcpy.implements_option_a = CPYFOptionA();

boolean rprivileged = if options<1> == '1' then AArch64.IsUnprivAccessPriv() else PSTATE.EL != EL0;
boolean wprivileged = if options<0> == '1' then AArch64.IsUnprivAccessPriv() else PSTATE.EL != EL0;

AccessDescriptor raccdesc = CreateAccDescMOPS(MemOp_LOAD, rprivileged, rnontemporal);
AccessDescriptor waccdesc = CreateAccDescMOPS(MemOp_STORE, wprivileged, wnontemporal);

if memcpy.stage == MOPSStage_Prologue then
    if memcpy.cpysize<63> == '1' then memcpy.cpysize = 0x7FFFFFFFFFFFFFFF;

    if memcpy.implements_option_a then
        memcpy.nzcv = '0000';
        // Copy in the forward direction offsets the arguments.
        memcpy.toaddress = memcpy.toaddress + memcpy.cpysize;
        memcpy.fromaddress = memcpy.fromaddress + memcpy.cpysize;
        memcpy.cpysize = 0 - memcpy.cpysize;
    else
        memcpy.nzcv = '0010';

memcpy.stagecpysize = MemCpyStageSize(memcpy);

if memcpy.stage != MOPSStage_Prologue then
    CheckMemCpyParams(memcpy, options);

integer copied;
boolean iswrite;
AddressDescriptor memaddrdesc;
PhysMemRetStatus memstatus;
memcpy.forward = TRUE;
boolean fault = FALSE;
integer B;

if memcpy.implements_option_a then
    while memcpy.stagecpysize != 0 && !fault do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(memcpy);
        assert B <= -1 * memcpy.stagecpysize;

        (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress + memcpy.cpysize,
memcpy.fromaddress + memcpy.cpysize, memcpy.forward, B, raccdesc, waccdesc);
        if copied != B then
            fault = TRUE;
        else
            memcpy.cpysize = memcpy.cpysize + B;
            memcpy.stagecpysize = memcpy.stagecpysize + B;
else
    while memcpy.stagecpysize > 0 && !fault do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(memcpy);
        assert B <= memcpy.stagecpysize;

        (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress, memcpy.fromaddress,
memcpy.forward, B, raccdesc, waccdesc);
        if copied != B then
            fault = TRUE;
        else
            memcpy.fromaddress = memcpy.fromaddress + B;
            memcpy.toaddress = memcpy.toaddress + B;
            memcpy.cpysize = memcpy.cpysize - B;
            memcpy.stagecpysize = memcpy.stagecpysize - B;

UpdateCpyRegisters(memcpy, fault, copied);
```

```
if fault then
    if IsFault(memaddrdesc) then
        AArch64.Abort(memaddrdesc.vaddress, memaddrdesc.fault);

    if IsFault(memstatus) then
        AccessDescriptor accdesc = if iswrite then waccdesc else raccdesc;
        HandleExternalAbort(memstatus, iswrite, memaddrdesc, B, accdesc);

if memcpy.stage == MOPSStage_Prologue then
    PSTATE.<N,Z,C,V> = memcpy.nzcv;
```

C6.2.83 CPYFPTWN, CPYFMTWN, CPYFETWN

Memory Copy Forward-only, reads and writes unprivileged, writes non-temporal. These instructions perform a memory copy. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYFPTWN, then CPYFMTWN, and then CPYFETWN.

CPYFPTWN performs some preconditioning of the arguments suitable for using the CPYFMTWN instruction, and performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYFMTWN performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYFETWN performs the last part of the memory copy.

Note

The inclusion of IMPLEMENTATION DEFINED amounts of memory copy allows some optimization of the size that can be performed.

The memory copy performed by these instructions is in the forward direction only, so the instructions are suitable for a memory copy only where there is no overlap between the source and destination locations, or where the source address is greater than the destination address.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

After execution of CPYFPTWN, option A (which results in encoding PSTATE.C = 0):

- If Xn<63> == 1, the copy size is saturated to `0x7FFFFFFFFFFFFFFF`.
- Xs holds the original Xs + saturated Xn.
- Xd holds the original Xd + saturated Xn.
- Xn holds -1* saturated Xn + an IMPLEMENTATION DEFINED number of bytes copied.
- PSTATE.{N,Z,V} are set to {0,0,0}.

After execution of CPYFPTWN, option B (which results in encoding PSTATE.C = 1):

- If Xn<63> == 1, the copy size is saturated to `0x7FFFFFFFFFFFFFFF`.
- Xs holds the original Xs + an IMPLEMENTATION DEFINED number of bytes copied.
- Xd holds the original Xd + an IMPLEMENTATION DEFINED number of bytes copied.
- Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
- PSTATE.{N,Z,V} are set to {0,0,0}.

For CPYFMTWN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number and holds -1* the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from -Xn.
- Xd holds the lowest address that the copy is made to -Xn.
- At the end of the instruction, the value of Xn is written back with -1* the number of bytes remaining to be copied in the memory copy in total.

For CPYFMTWN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from.

- Xd holds the lowest address that the copy is copied to.
- At the end of the instruction:
  - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
  - the value of Xs is written back with the lowest address that has not been copied from.
  - the value of Xd is written back with the lowest address that has not been copied to.

For CPYFETWN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

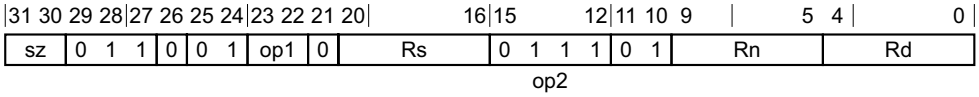
- Xn is treated as a signed 64-bit number and holds -1* the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from -Xn.
- Xd holds the lowest address that the copy is made to -Xn.
- At the end of the instruction, the value of Xn is written back with 0.

For CPYFETWN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from.
- Xd holds the lowest address that the copy is copied to.
- At the end of the instruction:
  - the value of Xn is written back with 0.
  - the value of Xs is written back with the lowest address that has not been copied from.
  - the value of Xd is written back with the lowest address that has not been copied to.

Integer

(FEAT_MOPS)



Epilogue variant

Applies when `op1 == 10`.

```
CPYFETWN [<Xd>]!, [<Xs>]!, <Xn>!
```

Main variant

Applies when `op1 == 01`.

```
CPYFMTWN [<Xd>]!, [<Xs>]!, <Xn>!
```

Prologue variant

Applies when `op1 == 00`.

```
CPYFPTWN [<Xd>]!, [<Xs>]!, <Xn>!
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_MOPS) || sz != '00' then UNDEFINED;

CPYParams memcpy;
memcpy.d = UInt(Rd);
memcpy.s = UInt(Rs);
memcpy.n = UInt(Rn);
bits(4) options = op2;
boolean rnontemporal = options<3> == '1';
boolean wnontemporal = options<2> == '1';

case op1 of
    when '00' memcpy.stage = MOPSStage_Prologue;
    when '01' memcpy.stage = MOPSStage_Main;
    when '10' memcpy.stage = MOPSStage_Epilogue;
    otherwise SEE "Memory Copy and Memory Set";

CheckMOPSEnabled();

if (memcpy.s == memcpy.n || memcpy.s == memcpy.d || memcpy.n == memcpy.d || memcpy.d == 31 || memcpy.s
== 31 || memcpy.n == 31) then
    Constraint c = ConstrainUnpredictable(Unpredictable_MOPSOVERLAP31);
    assert c IN {Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNDEF UNDEFINED;
        when Constraint_NOP   EndOfInstruction();
```

Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*, and particularly *Memory Copy and Memory Set CPY**.

Assembler symbols

<Xd> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the "Rd" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.

<Xs> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the "Rs" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the "Rs" field.

<Xn> For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero at the end of the instruction, encoded in the "Rn" field.

For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the "Rn" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the "Rn" field.

Operation

```
constant integer N = MaxBlockSizeCopiedBytes();
bits(8*N) readdata;

memcpy.nzcv = PSTATE.<N,Z,C,V>;
memcpy.toaddress = X[memcpy.d, 64];
```

```
memcpy.fromaddress = X[memcpy.s, 64];
memcpy.cpysize = SInt(X[memcpy.n, 64]);
memcpy.implements_option_a = CPYFOptionA();

boolean rprivileged = if options<1> == '1' then AArch64.IsUnprivAccessPriv() else PSTATE.EL != EL0;
boolean wprivileged = if options<0> == '1' then AArch64.IsUnprivAccessPriv() else PSTATE.EL != EL0;

AccessDescriptor raccdesc = CreateAccDescMOPS(MemOp_LOAD, rprivileged, rnontemporal);
AccessDescriptor waccdesc = CreateAccDescMOPS(MemOp_STORE, wprivileged, wnontemporal);

if memcpy.stage == MOPSStage_Prologue then
    if memcpy.cpysize<63> == '1' then memcpy.cpysize = 0x7FFFFFFFFFFFFFFF;

    if memcpy.implements_option_a then
        memcpy.nzcv = '0000';
        // Copy in the forward direction offsets the arguments.
        memcpy.toaddress = memcpy.toaddress + memcpy.cpysize;
        memcpy.fromaddress = memcpy.fromaddress + memcpy.cpysize;
        memcpy.cpysize = 0 - memcpy.cpysize;
    else
        memcpy.nzcv = '0010';

memcpy.stagecpysize = MemCpyStageSize(memcpy);

if memcpy.stage != MOPSStage_Prologue then
    CheckMemCpyParams(memcpy, options);

integer copied;
boolean iswrite;
AddressDescriptor memaddrdesc;
PhysMemRetStatus memstatus;
memcpy.forward = TRUE;
boolean fault = FALSE;
integer B;

if memcpy.implements_option_a then
    while memcpy.stagecpysize != 0 && !fault do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(memcpy);
        assert B <= -1 * memcpy.stagecpysize;

        (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress + memcpy.cpysize,
memcpy.fromaddress + memcpy.cpysize, memcpy.forward, B, raccdesc, waccdesc);
        if copied != B then
            fault = TRUE;
        else
            memcpy.cpysize = memcpy.cpysize + B;
            memcpy.stagecpysize = memcpy.stagecpysize + B;
else
    while memcpy.stagecpysize > 0 && !fault do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(memcpy);
        assert B <= memcpy.stagecpysize;

        (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress, memcpy.fromaddress,
memcpy.forward, B, raccdesc, waccdesc);
        if copied != B then
            fault = TRUE;
        else
            memcpy.fromaddress = memcpy.fromaddress + B;
            memcpy.toaddress = memcpy.toaddress + B;
            memcpy.cpysize = memcpy.cpysize - B;
            memcpy.stagecpysize = memcpy.stagecpysize - B;

UpdateCpyRegisters(memcpy, fault, copied);
```

```
if fault then
    if IsFault(memaddrdesc) then
        AArch64.Abort(memaddrdesc.vaddress, memaddrdesc.fault);

    if IsFault(memstatus) then
        AccessDescriptor accdesc = if iswrite then waccdesc else raccdesc;
        HandleExternalAbort(memstatus, iswrite, memaddrdesc, B, accdesc);

if memcpy.stage == MOPSStage_Prologue then
    PSTATE.<N,Z,C,V> = memcpy.nzcv;
```

C6.2.84 CPYFPWN, CPYFMWN, CPYFEWN

Memory Copy Forward-only, writes non-temporal. These instructions perform a memory copy. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYFPWN, then CPYFMWN, and then CPYFEWN.

CPYFPWN performs some preconditioning of the arguments suitable for using the CPYFMWN instruction, and performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYFMWN performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYFEWN performs the last part of the memory copy.

Note

The inclusion of IMPLEMENTATION DEFINED amounts of memory copy allows some optimization of the size that can be performed.

The memory copy performed by these instructions is in the forward direction only, so the instructions are suitable for a memory copy only where there is no overlap between the source and destination locations, or where the source address is greater than the destination address.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

After execution of CPYFPWN, option A (which results in encoding PSTATE.C = 0):

- If Xn<63> == 1, the copy size is saturated to `0x7FFFFFFFFFFFFFFF`.
- Xs holds the original Xs + saturated Xn.
- Xd holds the original Xd + saturated Xn.
- Xn holds -1* saturated Xn + an IMPLEMENTATION DEFINED number of bytes copied.
- PSTATE.{N,Z,V} are set to {0,0,0}.

After execution of CPYFPWN, option B (which results in encoding PSTATE.C = 1):

- If Xn<63> == 1, the copy size is saturated to `0x7FFFFFFFFFFFFFFF`.
- Xs holds the original Xs + an IMPLEMENTATION DEFINED number of bytes copied.
- Xd holds the original Xd + an IMPLEMENTATION DEFINED number of bytes copied.
- Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
- PSTATE.{N,Z,V} are set to {0,0,0}.

For CPYFMWN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number and holds -1* the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from -Xn.
- Xd holds the lowest address that the copy is made to -Xn.
- At the end of the instruction, the value of Xn is written back with -1* the number of bytes remaining to be copied in the memory copy in total.

For CPYFMWN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from.

- Xd holds the lowest address that the copy is copied to.
- At the end of the instruction:
  - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
  - the value of Xs is written back with the lowest address that has not been copied from.
  - the value of Xd is written back with the lowest address that has not been copied to.

For CPYFEWN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

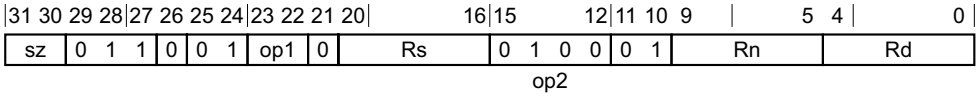
- Xn is treated as a signed 64-bit number and holds -1* the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from -Xn.
- Xd holds the lowest address that the copy is made to -Xn.
- At the end of the instruction, the value of Xn is written back with 0.

For CPYFEWN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from.
- Xd holds the lowest address that the copy is copied to.
- At the end of the instruction:
  - the value of Xn is written back with 0.
  - the value of Xs is written back with the lowest address that has not been copied from.
  - the value of Xd is written back with the lowest address that has not been copied to.

Integer

(FEAT_MOPS)



Epilogue variant

Applies when `op1 == 10`.

```
CPYFEWN [<Xd>]!, [<Xs>]!, <Xn>!
```

Main variant

Applies when `op1 == 01`.

```
CPYFMWN [<Xd>]!, [<Xs>]!, <Xn>!
```

Prologue variant

Applies when `op1 == 00`.

```
CPYFPWN [<Xd>]!, [<Xs>]!, <Xn>!
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_MOPS) || sz != '00' then UNDEFINED;

CPYParams memcpy;
memcpy.d = UInt(Rd);
memcpy.s = UInt(Rs);
memcpy.n = UInt(Rn);
bits(4) options = op2;
boolean rnontemporal = options<3> == '1';
boolean wnontemporal = options<2> == '1';

case op1 of
    when '00' memcpy.stage = MOPSStage_Prologue;
    when '01' memcpy.stage = MOPSStage_Main;
    when '10' memcpy.stage = MOPSStage_Epilogue;
    otherwise SEE "Memory Copy and Memory Set";

CheckMOPSEnabled();

if (memcpy.s == memcpy.n || memcpy.s == memcpy.d || memcpy.n == memcpy.d || memcpy.d == 31 || memcpy.s
== 31 || memcpy.n == 31) then
    Constraint c = ConstrainUnpredictable(Unpredictable_MOPSOVERLAP31);
    assert c IN {Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNDEF UNDEFINED;
        when Constraint_NOP   EndOfInstruction();
```

Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*, and particularly *Memory Copy and Memory Set CPY**.

Assembler symbols

<Xd> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the "Rd" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.

<Xs> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the "Rs" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the "Rs" field.

<Xn> For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero at the end of the instruction, encoded in the "Rn" field.

For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the "Rn" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the "Rn" field.

Operation

```
constant integer N = MaxBlockSizeCopiedBytes();
bits(8*N) readdata;

memcpy.nzcv = PSTATE.<N,Z,C,V>;
memcpy.toaddress = X[memcpy.d, 64];
```

```
memcpy.fromaddress = X[memcpy.s, 64];
memcpy.cpysize = SInt(X[memcpy.n, 64]);
memcpy.implements_option_a = CPYFOptionA();

boolean rprivileged = if options<1> == '1' then AArch64.IsUnprivAccessPriv() else PSTATE.EL != EL0;
boolean wprivileged = if options<0> == '1' then AArch64.IsUnprivAccessPriv() else PSTATE.EL != EL0;

AccessDescriptor raccdesc = CreateAccDescMOPS(MemOp_LOAD, rprivileged, rnontemporal);
AccessDescriptor waccdesc = CreateAccDescMOPS(MemOp_STORE, wprivileged, wnontemporal);

if memcpy.stage == MOPSStage_Prologue then
    if memcpy.cpysize<63> == '1' then memcpy.cpysize = 0x7FFFFFFFFFFFFFFF;

    if memcpy.implements_option_a then
        memcpy.nzcv = '0000';
        // Copy in the forward direction offsets the arguments.
        memcpy.toaddress = memcpy.toaddress + memcpy.cpysize;
        memcpy.fromaddress = memcpy.fromaddress + memcpy.cpysize;
        memcpy.cpysize = 0 - memcpy.cpysize;
    else
        memcpy.nzcv = '0010';

memcpy.stagecpysize = MemCpyStageSize(memcpy);

if memcpy.stage != MOPSStage_Prologue then
    CheckMemCpyParams(memcpy, options);

integer copied;
boolean iswrite;
AddressDescriptor memaddrdesc;
PhysMemRetStatus memstatus;
memcpy.forward = TRUE;
boolean fault = FALSE;
integer B;

if memcpy.implements_option_a then
    while memcpy.stagecpysize != 0 && !fault do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(memcpy);
        assert B <= -1 * memcpy.stagecpysize;

        (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress + memcpy.cpysize,
memcpy.fromaddress + memcpy.cpysize, memcpy.forward, B, raccdesc, waccdesc);
        if copied != B then
            fault = TRUE;
        else
            memcpy.cpysize = memcpy.cpysize + B;
            memcpy.stagecpysize = memcpy.stagecpysize + B;
else
    while memcpy.stagecpysize > 0 && !fault do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(memcpy);
        assert B <= memcpy.stagecpysize;

        (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress, memcpy.fromaddress,
memcpy.forward, B, raccdesc, waccdesc);
        if copied != B then
            fault = TRUE;
        else
            memcpy.fromaddress = memcpy.fromaddress + B;
            memcpy.toaddress = memcpy.toaddress + B;
            memcpy.cpysize = memcpy.cpysize - B;
            memcpy.stagecpysize = memcpy.stagecpysize - B;

UpdateCpyRegisters(memcpy, fault, copied);
```

```
if fault then
    if IsFault(memaddrdesc) then
        AArch64.Abort(memaddrdesc.vaddress, memaddrdesc.fault);

    if IsFault(memstatus) then
        AccessDescriptor accdesc = if iswrite then waccdesc else raccdesc;
        HandleExternalAbort(memstatus, iswrite, memaddrdesc, B, accdesc);

if memcpy.stage == MOPSStage_Prologue then
    PSTATE.<N,Z,C,V> = memcpy.nzcv;
```

C6.2.85 CPYFPWT, CPYFMWT, CPYFEWT

Memory Copy Forward-only, writes unprivileged. These instructions perform a memory copy. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYFPWT, then CPYFMWT, and then CPYFEWT.

CPYFPWT performs some preconditioning of the arguments suitable for using the CPYFMWT instruction, and performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYFMWT performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYFEWT performs the last part of the memory copy.

Note

The inclusion of IMPLEMENTATION DEFINED amounts of memory copy allows some optimization of the size that can be performed.

The memory copy performed by these instructions is in the forward direction only, so the instructions are suitable for a memory copy only where there is no overlap between the source and destination locations, or where the source address is greater than the destination address.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

After execution of CPYFPWT, option A (which results in encoding PSTATE.C = 0):

- If Xn<63> == 1, the copy size is saturated to `0x7FFFFFFFFFFFFFFF`.
- Xs holds the original Xs + saturated Xn.
- Xd holds the original Xd + saturated Xn.
- Xn holds -1* saturated Xn + an IMPLEMENTATION DEFINED number of bytes copied.
- PSTATE.{N,Z,V} are set to {0,0,0}.

After execution of CPYFPWT, option B (which results in encoding PSTATE.C = 1):

- If Xn<63> == 1, the copy size is saturated to `0x7FFFFFFFFFFFFFFF`.
- Xs holds the original Xs + an IMPLEMENTATION DEFINED number of bytes copied.
- Xd holds the original Xd + an IMPLEMENTATION DEFINED number of bytes copied.
- Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
- PSTATE.{N,Z,V} are set to {0,0,0}.

For CPYFMWT, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number and holds -1* the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from -Xn.
- Xd holds the lowest address that the copy is made to -Xn.
- At the end of the instruction, the value of Xn is written back with -1* the number of bytes remaining to be copied in the memory copy in total.

For CPYFMWT, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from.

- Xd holds the lowest address that the copy is copied to.
- At the end of the instruction:
  - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
  - the value of Xs is written back with the lowest address that has not been copied from.
  - the value of Xd is written back with the lowest address that has not been copied to.

For CPYFEWT, option A (encoded by PSTATE.C = 0), the format of the arguments is:

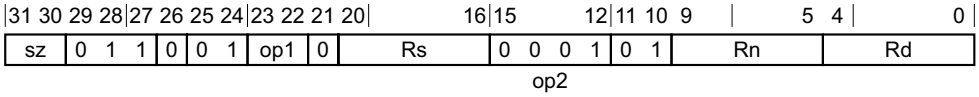
- Xn is treated as a signed 64-bit number and holds -1* the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from -Xn.
- Xd holds the lowest address that the copy is made to -Xn.
- At the end of the instruction, the value of Xn is written back with 0.

For CPYFEWT, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from.
- Xd holds the lowest address that the copy is copied to.
- At the end of the instruction:
  - the value of Xn is written back with 0.
  - the value of Xs is written back with the lowest address that has not been copied from.
  - the value of Xd is written back with the lowest address that has not been copied to.

Integer

(FEAT_MOPS)



Epilogue variant

Applies when `op1 == 10`.

```
CPYFEWT [<Xd>]!, [<Xs>]!, <Xn>!
```

Main variant

Applies when `op1 == 01`.

```
CPYFMWT [<Xd>]!, [<Xs>]!, <Xn>!
```

Prologue variant

Applies when `op1 == 00`.

```
CPYFPWT [<Xd>]!, [<Xs>]!, <Xn>!
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_MOPS) || sz != '00' then UNDEFINED;

CPYParams memcpy;
memcpy.d = UInt(Rd);
memcpy.s = UInt(Rs);
memcpy.n = UInt(Rn);
bits(4) options = op2;
boolean rnontemporal = options<3> == '1';
boolean wnontemporal = options<2> == '1';

case op1 of
    when '00' memcpy.stage = MOPSStage_Prologue;
    when '01' memcpy.stage = MOPSStage_Main;
    when '10' memcpy.stage = MOPSStage_Epilogue;
    otherwise SEE "Memory Copy and Memory Set";

CheckMOPSEnabled();

if (memcpy.s == memcpy.n || memcpy.s == memcpy.d || memcpy.n == memcpy.d || memcpy.d == 31 || memcpy.s
== 31 || memcpy.n == 31) then
    Constraint c = ConstrainUnpredictable(Unpredictable_MOPSOVERLAP31);
    assert c IN {Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNDEF UNDEFINED;
        when Constraint_NOP   EndOfInstruction();
```

Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*, and particularly *Memory Copy and Memory Set CPY**.

Assembler symbols

<Xd> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the "Rd" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.

<Xs> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the "Rs" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the "Rs" field.

<Xn> For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero at the end of the instruction, encoded in the "Rn" field.

For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the "Rn" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the "Rn" field.

Operation

```
constant integer N = MaxBlockSizeCopiedBytes();
bits(8*N) readdata;

memcpy.nzcv = PSTATE.<N,Z,C,V>;
memcpy.toaddress = X[memcpy.d, 64];
```

```
memcpy.fromaddress = X[memcpy.s, 64];
memcpy.cpysize = SInt(X[memcpy.n, 64]);
memcpy.implements_option_a = CPYFOptionA();

boolean rprivileged = if options<1> == '1' then AArch64.IsUnprivAccessPriv() else PSTATE.EL != EL0;
boolean wprivileged = if options<0> == '1' then AArch64.IsUnprivAccessPriv() else PSTATE.EL != EL0;

AccessDescriptor raccdesc = CreateAccDescMOPS(MemOp_LOAD, rprivileged, rnontemporal);
AccessDescriptor waccdesc = CreateAccDescMOPS(MemOp_STORE, wprivileged, wnontemporal);

if memcpy.stage == MOPSStage_Prologue then
    if memcpy.cpysize<63> == '1' then memcpy.cpysize = 0x7FFFFFFFFFFFFFFF;

    if memcpy.implements_option_a then
        memcpy.nzcv = '0000';
        // Copy in the forward direction offsets the arguments.
        memcpy.toaddress = memcpy.toaddress + memcpy.cpysize;
        memcpy.fromaddress = memcpy.fromaddress + memcpy.cpysize;
        memcpy.cpysize = 0 - memcpy.cpysize;
    else
        memcpy.nzcv = '0010';

memcpy.stagecpysize = MemCpyStageSize(memcpy);

if memcpy.stage != MOPSStage_Prologue then
    CheckMemCpyParams(memcpy, options);

integer copied;
boolean iswrite;
AddressDescriptor memaddrdesc;
PhysMemRetStatus memstatus;
memcpy.forward = TRUE;
boolean fault = FALSE;
integer B;

if memcpy.implements_option_a then
    while memcpy.stagecpysize != 0 && !fault do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(memcpy);
        assert B <= -1 * memcpy.stagecpysize;

        (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress + memcpy.cpysize,
memcpy.fromaddress + memcpy.cpysize, memcpy.forward, B, raccdesc, waccdesc);
        if copied != B then
            fault = TRUE;
        else
            memcpy.cpysize = memcpy.cpysize + B;
            memcpy.stagecpysize = memcpy.stagecpysize + B;
else
    while memcpy.stagecpysize > 0 && !fault do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(memcpy);
        assert B <= memcpy.stagecpysize;

        (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress, memcpy.fromaddress,
memcpy.forward, B, raccdesc, waccdesc);
        if copied != B then
            fault = TRUE;
        else
            memcpy.fromaddress = memcpy.fromaddress + B;
            memcpy.toaddress = memcpy.toaddress + B;
            memcpy.cpysize = memcpy.cpysize - B;
            memcpy.stagecpysize = memcpy.stagecpysize - B;

UpdateCpyRegisters(memcpy, fault, copied);
```

```
if fault then
    if IsFault(memaddrdesc) then
        AArch64.Abort(memaddrdesc.vaddress, memaddrdesc.fault);

    if IsFault(memstatus) then
        AccessDescriptor accdesc = if iswrite then waccdesc else raccdesc;
        HandleExternalAbort(memstatus, iswrite, memaddrdesc, B, accdesc);

if memcpy.stage == MOPSStage_Prologue then
    PSTATE.<N,Z,C,V> = memcpy.nzcv;
```

C6.2.86 CPYFPWTN, CPYFMWTN, CPYFEWTN

Memory Copy Forward-only, writes unprivileged, reads and writes non-temporal. These instructions perform a memory copy. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYFPWTN, then CPYFMWTN, and then CPYFEWTN.

CPYFPWTN performs some preconditioning of the arguments suitable for using the CPYFMWTN instruction, and performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYFMWTN performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYFEWTN performs the last part of the memory copy.

Note

The inclusion of IMPLEMENTATION DEFINED amounts of memory copy allows some optimization of the size that can be performed.

The memory copy performed by these instructions is in the forward direction only, so the instructions are suitable for a memory copy only where there is no overlap between the source and destination locations, or where the source address is greater than the destination address.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

After execution of CPYFPWTN, option A (which results in encoding PSTATE.C = 0):

- If Xn<63> == 1, the copy size is saturated to `0x7FFFFFFFFFFFFFFF`.
- Xs holds the original Xs + saturated Xn.
- Xd holds the original Xd + saturated Xn.
- Xn holds -1* saturated Xn + an IMPLEMENTATION DEFINED number of bytes copied.
- PSTATE.{N,Z,V} are set to {0,0,0}.

After execution of CPYFPWTN, option B (which results in encoding PSTATE.C = 1):

- If Xn<63> == 1, the copy size is saturated to `0x7FFFFFFFFFFFFFFF`.
- Xs holds the original Xs + an IMPLEMENTATION DEFINED number of bytes copied.
- Xd holds the original Xd + an IMPLEMENTATION DEFINED number of bytes copied.
- Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
- PSTATE.{N,Z,V} are set to {0,0,0}.

For CPYFMWTN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number and holds -1* the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from -Xn.
- Xd holds the lowest address that the copy is made to -Xn.
- At the end of the instruction, the value of Xn is written back with -1* the number of bytes remaining to be copied in the memory copy in total.

For CPYFMWTN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from.

- Xd holds the lowest address that the copy is copied to.
- At the end of the instruction:
  - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
  - the value of Xs is written back with the lowest address that has not been copied from.
  - the value of Xd is written back with the lowest address that has not been copied to.

For CPYFEWTN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

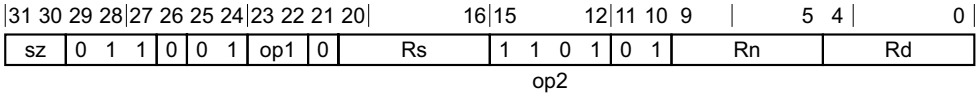
- Xn is treated as a signed 64-bit number and holds -1* the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from -Xn.
- Xd holds the lowest address that the copy is made to -Xn.
- At the end of the instruction, the value of Xn is written back with 0.

For CPYFEWTN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from.
- Xd holds the lowest address that the copy is copied to.
- At the end of the instruction:
  - the value of Xn is written back with 0.
  - the value of Xs is written back with the lowest address that has not been copied from.
  - the value of Xd is written back with the lowest address that has not been copied to.

Integer

(FEAT_MOPS)



Epilogue variant

Applies when `op1 == 10`.

```
CPYFEWTN [<Xd>]!, [<Xs>]!, <Xn>!
```

Main variant

Applies when `op1 == 01`.

```
CPYFMWTN [<Xd>]!, [<Xs>]!, <Xn>!
```

Prologue variant

Applies when `op1 == 00`.

```
CPYFPWTN [<Xd>]!, [<Xs>]!, <Xn>!
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_MOPS) || sz != '00' then UNDEFINED;

CPYParams memcpy;
memcpy.d = UInt(Rd);
memcpy.s = UInt(Rs);
memcpy.n = UInt(Rn);
bits(4) options = op2;
boolean rnontemporal = options<3> == '1';
boolean wnontemporal = options<2> == '1';

case op1 of
    when '00' memcpy.stage = MOPSStage_Prologue;
    when '01' memcpy.stage = MOPSStage_Main;
    when '10' memcpy.stage = MOPSStage_Epilogue;
    otherwise SEE "Memory Copy and Memory Set";

CheckMOPSEnabled();

if (memcpy.s == memcpy.n || memcpy.s == memcpy.d || memcpy.n == memcpy.d || memcpy.d == 31 || memcpy.s
== 31 || memcpy.n == 31) then
    Constraint c = ConstrainUnpredictable(Unpredictable_MOPSOVERLAP31);
    assert c IN {Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNDEF UNDEFINED;
        when Constraint_NOP   EndOfInstruction();
```

Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*, and particularly *Memory Copy and Memory Set CPY**.

Assembler symbols

<Xd> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the "Rd" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.

<Xs> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the "Rs" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the "Rs" field.

<Xn> For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero at the end of the instruction, encoded in the "Rn" field.

For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the "Rn" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the "Rn" field.

Operation

```
constant integer N = MaxBlockSizeCopiedBytes();
bits(8*N) readdata;

memcpy.nzcv = PSTATE.<N,Z,C,V>;
memcpy.toaddress = X[memcpy.d, 64];
```

```
memcpy.fromaddress = X[memcpy.s, 64];
memcpy.cpysize = SInt(X[memcpy.n, 64]);
memcpy.implements_option_a = CPYFOptionA();

boolean rprivileged = if options<1> == '1' then AArch64.IsUnprivAccessPriv() else PSTATE.EL != EL0;
boolean wprivileged = if options<0> == '1' then AArch64.IsUnprivAccessPriv() else PSTATE.EL != EL0;

AccessDescriptor raccdesc = CreateAccDescMOPS(MemOp_LOAD, rprivileged, rnontemporal);
AccessDescriptor waccdesc = CreateAccDescMOPS(MemOp_STORE, wprivileged, wnontemporal);

if memcpy.stage == MOPSStage_Prologue then
    if memcpy.cpysize<63> == '1' then memcpy.cpysize = 0x7FFFFFFFFFFFFFFF;

    if memcpy.implements_option_a then
        memcpy.nzcv = '0000';
        // Copy in the forward direction offsets the arguments.
        memcpy.toaddress = memcpy.toaddress + memcpy.cpysize;
        memcpy.fromaddress = memcpy.fromaddress + memcpy.cpysize;
        memcpy.cpysize = 0 - memcpy.cpysize;
    else
        memcpy.nzcv = '0010';

memcpy.stagecpysize = MemCpyStageSize(memcpy);

if memcpy.stage != MOPSStage_Prologue then
    CheckMemCpyParams(memcpy, options);

integer copied;
boolean iswrite;
AddressDescriptor memaddrdesc;
PhysMemRetStatus memstatus;
memcpy.forward = TRUE;
boolean fault = FALSE;
integer B;

if memcpy.implements_option_a then
    while memcpy.stagecpysize != 0 && !fault do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(memcpy);
        assert B <= -1 * memcpy.stagecpysize;

        (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress + memcpy.cpysize,
memcpy.fromaddress + memcpy.cpysize, memcpy.forward, B, raccdesc, waccdesc);
        if copied != B then
            fault = TRUE;
        else
            memcpy.cpysize = memcpy.cpysize + B;
            memcpy.stagecpysize = memcpy.stagecpysize + B;
else
    while memcpy.stagecpysize > 0 && !fault do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(memcpy);
        assert B <= memcpy.stagecpysize;

        (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress, memcpy.fromaddress,
memcpy.forward, B, raccdesc, waccdesc);
        if copied != B then
            fault = TRUE;
        else
            memcpy.fromaddress = memcpy.fromaddress + B;
            memcpy.toaddress = memcpy.toaddress + B;
            memcpy.cpysize = memcpy.cpysize - B;
            memcpy.stagecpysize = memcpy.stagecpysize - B;

UpdateCpyRegisters(memcpy, fault, copied);
```

```
if fault then
    if IsFault(memaddrdesc) then
        AArch64.Abort(memaddrdesc.vaddress, memaddrdesc.fault);

    if IsFault(memstatus) then
        AccessDescriptor accdesc = if iswrite then waccdesc else raccdesc;
        HandleExternalAbort(memstatus, iswrite, memaddrdesc, B, accdesc);

if memcpy.stage == MOPSStage_Prologue then
    PSTATE.<N,Z,C,V> = memcpy.nzcv;
```

C6.2.87 CPYFPWTRN, CPYFMWTRN, CPYFEWTRN

Memory Copy Forward-only, writes unprivileged, reads non-temporal. These instructions perform a memory copy. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYFPWTRN, then CPYFMWTRN, and then CPYFEWTRN.

CPYFPWTRN performs some preconditioning of the arguments suitable for using the CPYFMWTRN instruction, and performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYFMWTRN performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYFEWTRN performs the last part of the memory copy.

Note

The inclusion of IMPLEMENTATION DEFINED amounts of memory copy allows some optimization of the size that can be performed.

The memory copy performed by these instructions is in the forward direction only, so the instructions are suitable for a memory copy only where there is no overlap between the source and destination locations, or where the source address is greater than the destination address.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

After execution of CPYFPWTRN, option A (which results in encoding PSTATE.C = 0):

- If Xn<63> == 1, the copy size is saturated to `0x7FFFFFFFFFFFFFFF`.
- Xs holds the original Xs + saturated Xn.
- Xd holds the original Xd + saturated Xn.
- Xn holds -1* saturated Xn + an IMPLEMENTATION DEFINED number of bytes copied.
- PSTATE.{N,Z,V} are set to {0,0,0}.

After execution of CPYFPWTRN, option B (which results in encoding PSTATE.C = 1):

- If Xn<63> == 1, the copy size is saturated to `0x7FFFFFFFFFFFFFFF`.
- Xs holds the original Xs + an IMPLEMENTATION DEFINED number of bytes copied.
- Xd holds the original Xd + an IMPLEMENTATION DEFINED number of bytes copied.
- Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
- PSTATE.{N,Z,V} are set to {0,0,0}.

For CPYFMWTRN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number and holds -1* the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from -Xn.
- Xd holds the lowest address that the copy is made to -Xn.
- At the end of the instruction, the value of Xn is written back with -1* the number of bytes remaining to be copied in the memory copy in total.

For CPYFMWTRN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from.

- Xd holds the lowest address that the copy is copied to.
- At the end of the instruction:
  - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
  - the value of Xs is written back with the lowest address that has not been copied from.
  - the value of Xd is written back with the lowest address that has not been copied to.

For CPYFEWTRN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

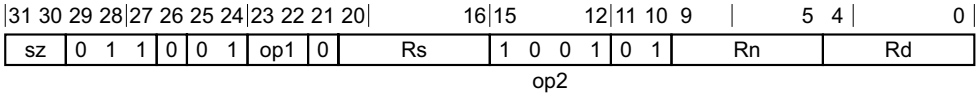
- Xn is treated as a signed 64-bit number and holds -1* the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from -Xn.
- Xd holds the lowest address that the copy is made to -Xn.
- At the end of the instruction, the value of Xn is written back with 0.

For CPYFEWTRN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from.
- Xd holds the lowest address that the copy is copied to.
- At the end of the instruction:
  - the value of Xn is written back with 0.
  - the value of Xs is written back with the lowest address that has not been copied from.
  - the value of Xd is written back with the lowest address that has not been copied to.

Integer

(FEAT_MOPS)



Epilogue variant

Applies when `op1 == 10`.

```
CPYFEWTRN [<Xd>]!, [<Xs>]!, <Xn>!
```

Main variant

Applies when `op1 == 01`.

```
CPYFMWTRN [<Xd>]!, [<Xs>]!, <Xn>!
```

Prologue variant

Applies when `op1 == 00`.

```
CPYFPWTRN [<Xd>]!, [<Xs>]!, <Xn>!
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_MOPS) || sz != '00' then UNDEFINED;

CPYParams memcpy;
memcpy.d = UInt(Rd);
memcpy.s = UInt(Rs);
memcpy.n = UInt(Rn);
bits(4) options = op2;
boolean rnontemporal = options<3> == '1';
boolean wnontemporal = options<2> == '1';

case op1 of
    when '00' memcpy.stage = MOPSStage_Prologue;
    when '01' memcpy.stage = MOPSStage_Main;
    when '10' memcpy.stage = MOPSStage_Epilogue;
    otherwise SEE "Memory Copy and Memory Set";

CheckMOPSEnabled();

if (memcpy.s == memcpy.n || memcpy.s == memcpy.d || memcpy.n == memcpy.d || memcpy.d == 31 || memcpy.s
== 31 || memcpy.n == 31) then
    Constraint c = ConstrainUnpredictable(Unpredictable_MOPSOVERLAP31);
    assert c IN {Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNDEF UNDEFINED;
        when Constraint_NOP   EndOfInstruction();
```

Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*, and particularly *Memory Copy and Memory Set CPY**.

Assembler symbols

<Xd> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the "Rd" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.

<Xs> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the "Rs" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the "Rs" field.

<Xn> For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero at the end of the instruction, encoded in the "Rn" field.

For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the "Rn" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the "Rn" field.

Operation

```
constant integer N = MaxBlockSizeCopiedBytes();
bits(8*N) readdata;

memcpy.nzcv = PSTATE.<N,Z,C,V>;
memcpy.toaddress = X[memcpy.d, 64];
```

```
memcpy.fromaddress = X[memcpy.s, 64];
memcpy.cpysize = SInt(X[memcpy.n, 64]);
memcpy.implements_option_a = CPYFOptionA();

boolean rprivileged = if options<1> == '1' then AArch64.IsUnprivAccessPriv() else PSTATE.EL != EL0;
boolean wprivileged = if options<0> == '1' then AArch64.IsUnprivAccessPriv() else PSTATE.EL != EL0;

AccessDescriptor raccdesc = CreateAccDescMOPS(MemOp_LOAD, rprivileged, rnontemporal);
AccessDescriptor waccdesc = CreateAccDescMOPS(MemOp_STORE, wprivileged, wnontemporal);

if memcpy.stage == MOPSStage_Prologue then
    if memcpy.cpysize<63> == '1' then memcpy.cpysize = 0x7FFFFFFFFFFFFFFF;

    if memcpy.implements_option_a then
        memcpy.nzcv = '0000';
        // Copy in the forward direction offsets the arguments.
        memcpy.toaddress = memcpy.toaddress + memcpy.cpysize;
        memcpy.fromaddress = memcpy.fromaddress + memcpy.cpysize;
        memcpy.cpysize = 0 - memcpy.cpysize;
    else
        memcpy.nzcv = '0010';

memcpy.stagecpysize = MemCpyStageSize(memcpy);

if memcpy.stage != MOPSStage_Prologue then
    CheckMemCpyParams(memcpy, options);

integer copied;
boolean iswrite;
AddressDescriptor memaddrdesc;
PhysMemRetStatus memstatus;
memcpy.forward = TRUE;
boolean fault = FALSE;
integer B;

if memcpy.implements_option_a then
    while memcpy.stagecpysize != 0 && !fault do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(memcpy);
        assert B <= -1 * memcpy.stagecpysize;

        (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress + memcpy.cpysize,
memcpy.fromaddress + memcpy.cpysize, memcpy.forward, B, raccdesc, waccdesc);
        if copied != B then
            fault = TRUE;
        else
            memcpy.cpysize = memcpy.cpysize + B;
            memcpy.stagecpysize = memcpy.stagecpysize + B;
else
    while memcpy.stagecpysize > 0 && !fault do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(memcpy);
        assert B <= memcpy.stagecpysize;

        (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress, memcpy.fromaddress,
memcpy.forward, B, raccdesc, waccdesc);
        if copied != B then
            fault = TRUE;
        else
            memcpy.fromaddress = memcpy.fromaddress + B;
            memcpy.toaddress = memcpy.toaddress + B;
            memcpy.cpysize = memcpy.cpysize - B;
            memcpy.stagecpysize = memcpy.stagecpysize - B;

UpdateCpyRegisters(memcpy, fault, copied);
```

```
if fault then
    if IsFault(memaddrdesc) then
        AArch64.Abort(memaddrdesc.vaddress, memaddrdesc.fault);

    if IsFault(memstatus) then
        AccessDescriptor accdesc = if iswrite then waccdesc else raccdesc;
        HandleExternalAbort(memstatus, iswrite, memaddrdesc, B, accdesc);

if memcpy.stage == MOPSStage_Prologue then
    PSTATE.<N,Z,C,V> = memcpy.nzcv;
```

C6.2.88 CPYFPWTWN, CPYFMWTWN, CPYFEWTWN

Memory Copy Forward-only, writes unprivileged and non-temporal. These instructions perform a memory copy. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYFPWTWN, then CPYFMWTWN, and then CPYFEWTWN.

CPYFPWTWN performs some preconditioning of the arguments suitable for using the CPYFMWTWN instruction, and performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYFMWTWN performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYFEWTWN performs the last part of the memory copy.

Note

The inclusion of IMPLEMENTATION DEFINED amounts of memory copy allows some optimization of the size that can be performed.

The memory copy performed by these instructions is in the forward direction only, so the instructions are suitable for a memory copy only where there is no overlap between the source and destination locations, or where the source address is greater than the destination address.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

After execution of CPYFPWTWN, option A (which results in encoding PSTATE.C = 0):

- If Xn<63> == 1, the copy size is saturated to `0x7FFFFFFFFFFFFFFF`.
- Xs holds the original Xs + saturated Xn.
- Xd holds the original Xd + saturated Xn.
- Xn holds -1* saturated Xn + an IMPLEMENTATION DEFINED number of bytes copied.
- PSTATE.{N,Z,V} are set to {0,0,0}.

After execution of CPYFPWTWN, option B (which results in encoding PSTATE.C = 1):

- If Xn<63> == 1, the copy size is saturated to `0x7FFFFFFFFFFFFFFF`.
- Xs holds the original Xs + an IMPLEMENTATION DEFINED number of bytes copied.
- Xd holds the original Xd + an IMPLEMENTATION DEFINED number of bytes copied.
- Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
- PSTATE.{N,Z,V} are set to {0,0,0}.

For CPYFMWTWN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number and holds -1* the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from -Xn.
- Xd holds the lowest address that the copy is made to -Xn.
- At the end of the instruction, the value of Xn is written back with -1* the number of bytes remaining to be copied in the memory copy in total.

For CPYFMWTWN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from.

- Xd holds the lowest address that the copy is copied to.
- At the end of the instruction:
  - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
  - the value of Xs is written back with the lowest address that has not been copied from.
  - the value of Xd is written back with the lowest address that has not been copied to.

For CPYFEWTWN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

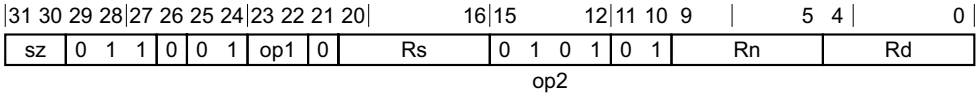
- Xn is treated as a signed 64-bit number and holds -1* the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from -Xn.
- Xd holds the lowest address that the copy is made to -Xn.
- At the end of the instruction, the value of Xn is written back with 0.

For CPYFEWTWN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from.
- Xd holds the lowest address that the copy is copied to.
- At the end of the instruction:
  - the value of Xn is written back with 0.
  - the value of Xs is written back with the lowest address that has not been copied from.
  - the value of Xd is written back with the lowest address that has not been copied to.

Integer

(FEAT_MOPS)



Epilogue variant

Applies when `op1 == 10`.

```
CPYFEWTWN [<Xd>]!, [<Xs>]!, <Xn>!
```

Main variant

Applies when `op1 == 01`.

```
CPYFMWTWN [<Xd>]!, [<Xs>]!, <Xn>!
```

Prologue variant

Applies when `op1 == 00`.

```
CPYFPWTWN [<Xd>]!, [<Xs>]!, <Xn>!
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_MOPS) || sz != '00' then UNDEFINED;

CPYParams memcpy;
memcpy.d = UInt(Rd);
memcpy.s = UInt(Rs);
memcpy.n = UInt(Rn);
bits(4) options = op2;
boolean rnontemporal = options<3> == '1';
boolean wnontemporal = options<2> == '1';

case op1 of
    when '00' memcpy.stage = MOPSStage_Prologue;
    when '01' memcpy.stage = MOPSStage_Main;
    when '10' memcpy.stage = MOPSStage_Epilogue;
    otherwise SEE "Memory Copy and Memory Set";

CheckMOPSEnabled();

if (memcpy.s == memcpy.n || memcpy.s == memcpy.d || memcpy.n == memcpy.d || memcpy.d == 31 || memcpy.s
== 31 || memcpy.n == 31) then
    Constraint c = ConstrainUnpredictable(Unpredictable_MOPSOVERLAP31);
    assert c IN {Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNDEF UNDEFINED;
        when Constraint_NOP   EndOfInstruction();
```

Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*, and particularly *Memory Copy and Memory Set CPY**.

Assembler symbols

<Xd> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the "Rd" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.

<Xs> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the "Rs" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the "Rs" field.

<Xn> For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero at the end of the instruction, encoded in the "Rn" field.

For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the "Rn" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the "Rn" field.

Operation

```
constant integer N = MaxBlockSizeCopiedBytes();
bits(8*N) readdata;

memcpy.nzcv = PSTATE.<N,Z,C,V>;
memcpy.toaddress = X[memcpy.d, 64];
```

```
memcpy.fromaddress = X[memcpy.s, 64];
memcpy.cpysize = SInt(X[memcpy.n, 64]);
memcpy.implements_option_a = CPYFOptionA();

boolean rprivileged = if options<1> == '1' then AArch64.IsUnprivAccessPriv() else PSTATE.EL != EL0;
boolean wprivileged = if options<0> == '1' then AArch64.IsUnprivAccessPriv() else PSTATE.EL != EL0;

AccessDescriptor raccdesc = CreateAccDescMOPS(MemOp_LOAD, rprivileged, rnontemporal);
AccessDescriptor waccdesc = CreateAccDescMOPS(MemOp_STORE, wprivileged, wnontemporal);

if memcpy.stage == MOPSStage_Prologue then
    if memcpy.cpysize<63> == '1' then memcpy.cpysize = 0x7FFFFFFFFFFFFFFF;

    if memcpy.implements_option_a then
        memcpy.nzcv = '0000';
        // Copy in the forward direction offsets the arguments.
        memcpy.toaddress = memcpy.toaddress + memcpy.cpysize;
        memcpy.fromaddress = memcpy.fromaddress + memcpy.cpysize;
        memcpy.cpysize = 0 - memcpy.cpysize;
    else
        memcpy.nzcv = '0010';

memcpy.stagecpysize = MemCpyStageSize(memcpy);

if memcpy.stage != MOPSStage_Prologue then
    CheckMemCpyParams(memcpy, options);

integer copied;
boolean iswrite;
AddressDescriptor memaddrdesc;
PhysMemRetStatus memstatus;
memcpy.forward = TRUE;
boolean fault = FALSE;
integer B;

if memcpy.implements_option_a then
    while memcpy.stagecpysize != 0 && !fault do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(memcpy);
        assert B <= -1 * memcpy.stagecpysize;

        (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress + memcpy.cpysize,
memcpy.fromaddress + memcpy.cpysize, memcpy.forward, B, raccdesc, waccdesc);
        if copied != B then
            fault = TRUE;
        else
            memcpy.cpysize = memcpy.cpysize + B;
            memcpy.stagecpysize = memcpy.stagecpysize + B;
else
    while memcpy.stagecpysize > 0 && !fault do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(memcpy);
        assert B <= memcpy.stagecpysize;

        (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress, memcpy.fromaddress,
memcpy.forward, B, raccdesc, waccdesc);
        if copied != B then
            fault = TRUE;
        else
            memcpy.fromaddress = memcpy.fromaddress + B;
            memcpy.toaddress = memcpy.toaddress + B;
            memcpy.cpysize = memcpy.cpysize - B;
            memcpy.stagecpysize = memcpy.stagecpysize - B;

UpdateCpyRegisters(memcpy, fault, copied);
```

```
if fault then
    if IsFault(memaddrdesc) then
        AArch64.Abort(memaddrdesc.vaddress, memaddrdesc.fault);

    if IsFault(memstatus) then
        AccessDescriptor accdesc = if iswrite then waccdesc else raccdesc;
        HandleExternalAbort(memstatus, iswrite, memaddrdesc, B, accdesc);

if memcpy.stage == MOPSStage_Prologue then
    PSTATE.<N,Z,C,V> = memcpy.nzcv;
```

C6.2.89 CPYP, CPYM, CPYE

Memory Copy. These instructions perform a memory copy. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYP, then CPYM, and then CPYE.

CPYP performs some preconditioning of the arguments suitable for using the CPYM instruction, and performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYM performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYE performs the last part of the memory copy.

Note

The inclusion of IMPLEMENTATION DEFINED amounts of memory copy allows some optimization of the size that can be performed.

For CPYP, the following saturation logic is applied:

If Xn<63:55> != 000000000, the copy size Xn is saturated to `0x007FFFFFFFFFFFF`.

After that saturation logic is applied, the direction of the memory copy is based on the following algorithm:

If (Xs > Xd) && (Xd + saturated Xn) > Xs, then direction = forward

Elsif (Xs < Xd) && (Xs + saturated Xn) > Xd, then direction = backward

Else direction = IMPLEMENTATION DEFINED choice between forward and backward.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

After execution of CPYP, option A (which results in encoding PSTATE.C = 0):

- PSTATE.{N,Z,V} are set to {0,0,0}.
- If the copy is in the forward direction, then:
  - Xs holds the original Xs + saturated Xn.
  - Xd holds the original Xd + saturated Xn.
  - Xn holds -1* saturated Xn + an IMPLEMENTATION DEFINED number of bytes copied.
- If the copy is in the backward direction, then:
  - Xs and Xd are unchanged.
  - Xn holds the saturated value of Xn - an IMPLEMENTATION DEFINED number of bytes copied.

After execution of CPYP, option B (which results in encoding PSTATE.C = 1):

- If the copy is in the forward direction, then:
  - Xs holds the original Xs + an IMPLEMENTATION DEFINED number of bytes copied.
  - Xd holds the original Xd + an IMPLEMENTATION DEFINED number of bytes copied.
  - Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - PSTATE.{N,Z,V} are set to {0,0,0}.
- If the copy is in the backward direction, then:
  - Xs holds the original Xs + saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - Xd holds the original Xd + saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - PSTATE.{N,Z,V} are set to {1,0,0}.

For CPYM, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number.
- If the copy is in the forward direction (Xn is a negative number), then:
  - Xn holds -1* the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the lowest address that the copy is copied from -Xn.
  - Xd holds the lowest address that the copy is copied to -Xn.
  - At the end of the instruction, the value of Xn is written back with -1* the number of bytes remaining to be copied in the memory copy in total.
- If the copy is in the backward direction (Xn is a positive number), then:
  - Xn holds the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the highest address that the copy is copied from -Xn+1.
  - Xd holds the highest address that the copy is copied to -Xn+1.
  - At the end of the instruction, the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.

For CPYM, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes to be copied in the memory copy in total.
- If the copy is in the forward direction (PSTATE.N == 0), then:
  - Xs holds the lowest address that the copy is copied from.
  - Xd holds the lowest address that the copy is copied to.
  - At the end of the instruction:
    - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
    - the value of Xs is written back with the lowest address that has not been copied from.
    - the value of Xd is written back with the lowest address that has not been copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
  - Xs holds the highest address that the copy is copied from +1.
  - Xd holds the highest address that the copy is copied to +1.
  - At the end of the instruction:
    - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
    - the value of Xs is written back with the highest address that has not been copied from +1.
    - the value of Xd is written back with the highest address that has not been copied to +1.

For CPYE, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number.
- If the copy is in the forward direction (Xn is a negative number), then:
  - Xn holds -1* the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the lowest address that the copy is copied from.
  - Xd holds the lowest address that the copy is made to.
  - At the end of the instruction, the value of Xn is written back with 0.
- If the copy is in the backward direction (Xn is a positive number), then:
  - Xn holds the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the highest address that the copy is copied from -Xn+1.

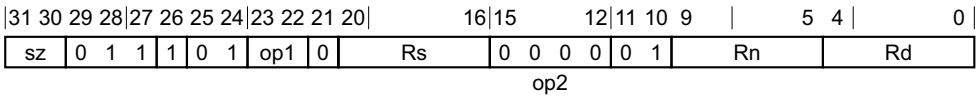
- Xd holds the highest address that the copy is copied to -Xn+1.
- At the end of the instruction, the value of Xn is written back with 0.

For CPYE, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes to be copied in the memory copy in total.
- If the copy is in the forward direction (PSTATE.N == 0), then:
  - Xs holds the lowest address that the copy is copied from.
  - Xd holds the lowest address that the copy is copied to.
  - At the end of the instruction:
    - the value of Xn is written back with 0.
    - the value of Xs is written back with the lowest address that has not been copied from.
    - the value of Xd is written back with the lowest address that has not been copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
  - Xs holds the highest address that the copy is copied from.
  - Xd holds the highest address that the copy is copied to.
  - At the end of the instruction:
    - the value of Xn is written back with 0.
    - the value of Xs is written back with the highest address that has not been copied from +1.
    - the value of Xd is written back with the highest address that has not been copied to +1.

Integer

(FEAT_MOPS)



Epilogue variant

Applies when `op1 == 10`.

```
CPYE [<Xd>]!, [<Xs>]!, <Xn>!
```

Main variant

Applies when `op1 == 01`.

```
CPYM [<Xd>]!, [<Xs>]!, <Xn>!
```

Prologue variant

Applies when `op1 == 00`.

```
CPYP [<Xd>]!, [<Xs>]!, <Xn>!
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_MOPS) || sz != '00' then UNDEFINED;

CPYParams memcpy;
memcpy.d = UInt(Rd);
memcpy.s = UInt(Rs);
memcpy.n = UInt(Rn);
bits(4) options = op2;
```

```
boolean rnontemporal = options<3> == '1';
boolean wnontemporal = options<2> == '1';

case op1 of
    when '00' memcpy.stage = MOPSStage_Prologue;
    when '01' memcpy.stage = MOPSStage_Main;
    when '10' memcpy.stage = MOPSStage_Epilogue;
    otherwise SEE "Memory Copy and Memory Set";

CheckMOPSEnabled();

if (memcpy.s == memcpy.n || memcpy.s == memcpy.d || memcpy.n == memcpy.d || memcpy.d == 31 || memcpy.s
== 31 || memcpy.n == 31) then
    Constraint c = ConstrainUnpredictable(Unpredictable_MOPSOVERLAP31);
    assert c IN {Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNDEF UNDEFINED;
        when Constraint_NOP   EndOfInstruction();
```

Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*, and particularly *Memory Copy and Memory Set CPY**.

Assembler symbols

<Xd> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the "Rd" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.

<Xs> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the "Rs" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the "Rs" field.

<Xn> For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero at the end of the instruction, encoded in the "Rn" field.

For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the "Rn" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the "Rn" field.

Operation

```
constant integer N = MaxBlockSizeCopiedBytes();
bits(8*N) readdata;

memcpy.nzcv = PSTATE.<N,Z,C,V>;
memcpy.toaddress = X[memcpy.d, 64];
memcpy.fromaddress = X[memcpy.s, 64];

if memcpy.stage == MOPSStage_Prologue then
    memcpy.cpysize = UInt(X[memcpy.n, 64]);
else
    memcpy.cpysize = SInt(X[memcpy.n, 64]);

memcpy.implements_option_a = CPYOptionA();
```

```
boolean rprivileged = if options<1> == '1' then AArch64.IsUnprivAccessPriv() else PSTATE.EL != EL0;
boolean wprivileged = if options<0> == '1' then AArch64.IsUnprivAccessPriv() else PSTATE.EL != EL0;

AccessDescriptor raccdesc = CreateAccDescMOPS(MemOp_LOAD, rprivileged, rnontemporal);
AccessDescriptor waccdesc = CreateAccDescMOPS(MemOp_STORE, wprivileged, wnontemporal);

if memcpy.stage == MOPSStage_Prologue then
    if memcpy.cpysize > 0x007FFFFFFFFFFFF then
        memcpy.cpysize = 0x007FFFFFFFFFFFF;

    memcpy.forward = IsMemCpyForward(memcpy);

    if memcpy.implements_option_a then
        memcpy.nzcv = '0000';
        if memcpy.forward then
            // Copy in the forward direction offsets the arguments.
            memcpy.toaddress = memcpy.toaddress + memcpy.cpysize;
            memcpy.fromaddress = memcpy.fromaddress + memcpy.cpysize;
            memcpy.cpysize = 0 - memcpy.cpysize;
    else
        if !memcpy.forward then
            // Copy in the reverse direction offsets the arguments.
            memcpy.toaddress = memcpy.toaddress + memcpy.cpysize;
            memcpy.fromaddress = memcpy.fromaddress + memcpy.cpysize;
            memcpy.nzcv = '1010';
        else
            memcpy.nzcv = '0010';

memcpy.stagecpysize = MemCpyStageSize(memcpy);

if memcpy.stage != MOPSStage_Prologue then
    memcpy.forward = memcpy.cpysize < 0 || (!memcpy.implements_option_a && memcpy.nzcv<3> == '0');
    CheckMemCpyParams(memcpy, options);

integer copied;
boolean iswrite;
AddressDescriptor memaddrdesc;
PhysMemRetStatus memstatus;
boolean fault = FALSE;
integer B;

if memcpy.implements_option_a then
    while memcpy.stagecpysize != 0 && !fault do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(memcpy);

        if memcpy.forward then
            assert B <= -1 * memcpy.stagecpysize;
            (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress + memcpy.cpysize,
memcpy.fromaddress + memcpy.cpysize, memcpy.forward, B, raccdesc, waccdesc);
            if copied != B then
                fault = TRUE;
            else
                memcpy.cpysize = memcpy.cpysize + B;
                memcpy.stagecpysize = memcpy.stagecpysize + B;

        else
            assert B <= memcpy.stagecpysize;
            memcpy.cpysize = memcpy.cpysize - B;
            memcpy.stagecpysize = memcpy.stagecpysize - B;

            (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress + memcpy.cpysize,
memcpy.fromaddress + memcpy.cpysize, memcpy.forward, B, raccdesc, waccdesc);
            if copied != B then
                fault = TRUE;
                memcpy.cpysize = memcpy.cpysize + B;
                memcpy.stagecpysize = memcpy.stagecpysize + B;
```

```
else
    while memcpy.stagecpysize > 0 && !fault do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(memcpy);
        assert B <= memcpy.stagecpysize;

        if memcpy.forward then
            (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress,
memcpy.fromaddress, memcpy.forward, B, raccdesc, waccdesc);
            if copied != B then
                fault = TRUE;
            else
                memcpy.fromaddress = memcpy.fromaddress + B;
                memcpy.toaddress = memcpy.toaddress + B;
        else
            (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress - B,
memcpy.fromaddress - B, memcpy.forward, B, raccdesc, waccdesc);

            if copied != B then
                fault = TRUE;
            else
                memcpy.fromaddress = memcpy.fromaddress - B;
                memcpy.toaddress = memcpy.toaddress - B;

        if !fault then
            memcpy.cpysize = memcpy.cpysize - B;
            memcpy.stagecpysize = memcpy.stagecpysize - B;

UpdateCpyRegisters(memcpy, fault, copied);

if fault then
    if IsFault(memaddrdesc) then
        AArch64.Abort(memaddrdesc.vaddress, memaddrdesc.fault);
    else
        AccessDescriptor accdesc = if iswrite then waccdesc else raccdesc;
        HandleExternalAbort(memstatus, iswrite, memaddrdesc, B, accdesc);

if memcpy.stage == MOPSStage_Prologue then
    PSTATE.<N,Z,C,V> = memcpy.nzcv;
```

C6.2.90 CPYPN, CPYMN, CPYEN

Memory Copy, reads and writes non-temporal. These instructions perform a memory copy. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYPN, then CPYMN, and then CPYEN.

CPYPN performs some preconditioning of the arguments suitable for using the CPYMN instruction, and performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYMN performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYEN performs the last part of the memory copy.

Note

The inclusion of IMPLEMENTATION DEFINED amounts of memory copy allows some optimization of the size that can be performed.

For CPYPN, the following saturation logic is applied:

If Xn<63:55> != 000000000, the copy size Xn is saturated to `0x007FFFFFFFFFFFFF`.

After that saturation logic is applied, the direction of the memory copy is based on the following algorithm:

If (Xs > Xd) && (Xd + saturated Xn) > Xs, then direction = forward

Elsif (Xs < Xd) && (Xs + saturated Xn) > Xd, then direction = backward

Else direction = IMPLEMENTATION DEFINED choice between forward and backward.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

After execution of CPYPN, option A (which results in encoding PSTATE.C = 0):

- PSTATE.{N,Z,V} are set to {0,0,0}.
- If the copy is in the forward direction, then:
  - Xs holds the original Xs + saturated Xn.
  - Xd holds the original Xd + saturated Xn.
  - Xn holds -1* saturated Xn + an IMPLEMENTATION DEFINED number of bytes copied.
- If the copy is in the backward direction, then:
  - Xs and Xd are unchanged.
  - Xn holds the saturated value of Xn - an IMPLEMENTATION DEFINED number of bytes copied.

After execution of CPYPN, option B (which results in encoding PSTATE.C = 1):

- If the copy is in the forward direction, then:
  - Xs holds the original Xs + an IMPLEMENTATION DEFINED number of bytes copied.
  - Xd holds the original Xd + an IMPLEMENTATION DEFINED number of bytes copied.
  - Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - PSTATE.{N,Z,V} are set to {0,0,0}.
- If the copy is in the backward direction, then:
  - Xs holds the original Xs + saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - Xd holds the original Xd + saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - PSTATE.{N,Z,V} are set to {1,0,0}.

For CPYMN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number.
- If the copy is in the forward direction (Xn is a negative number), then:
  - Xn holds -1* the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the lowest address that the copy is copied from -Xn.
  - Xd holds the lowest address that the copy is made to -Xn.
  - At the end of the instruction, the value of Xn is written back with -1* the number of bytes remaining to be copied in the memory copy in total.
- If the copy is in the backward direction (Xn is a positive number), then:
  - Xn holds the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the highest address that the copy is copied from -Xn+1.
  - Xd holds the highest address that the copy is copied to -Xn+1.
  - At the end of the instruction, the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.

For CPYMN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes to be copied in the memory copy in total.
- If the copy is in the forward direction (PSTATE.N == 0), then:
  - Xs holds the lowest address that the copy is copied from.
  - Xd holds the lowest address that the copy is copied to.
  - At the end of the instruction:
    - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
    - the value of Xs is written back with the lowest address that has not been copied from.
    - the value of Xd is written back with the lowest address that has not been copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
  - Xs holds the highest address that the copy is copied from +1.
  - Xd holds the highest address that the copy is copied to +1.
  - At the end of the instruction:
    - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
    - the value of Xs is written back with the highest address that has not been copied from +1.
    - the value of Xd is written back with the highest address that has not been copied to +1.

For CPYEN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number.
- If the copy is in the forward direction (Xn is a negative number), then:
  - Xn holds -1* the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the lowest address that the copy is copied from -Xn.
  - Xd holds the lowest address that the copy is made to -Xn.
  - At the end of the instruction, the value of Xn is written back with 0.
- If the copy is in the backward direction (Xn is a positive number), then:
  - Xn holds the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the highest address that the copy is copied from -Xn+1.

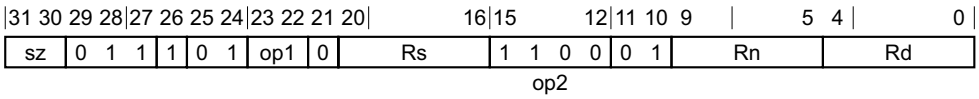
  - Xd holds the highest address that the copy is copied to -Xn+1.
  - At the end of the instruction, the value of Xn is written back with 0.

For CPYEN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes to be copied in the memory copy in total.
- If the copy is in the forward direction (PSTATE.N == 0), then:
  - Xs holds the lowest address that the copy is copied from.
  - Xd holds the lowest address that the copy is copied to.
  - At the end of the instruction:
    - the value of Xn is written back with 0.
    - the value of Xs is written back with the lowest address that has not been copied from.
    - the value of Xd is written back with the lowest address that has not been copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
  - Xs holds the highest address that the copy is copied from +1.
  - Xd holds the highest address that the copy is copied to +1.
  - At the end of the instruction:
    - the value of Xn is written back with 0.
    - the value of Xs is written back with the highest address that has not been copied from +1.
    - the value of Xd is written back with the highest address that has not been copied to +1.

Integer

(FEAT_MOPS)



Epilogue variant

Applies when `op1 == 10`.

```
CPYEN [<Xd>]!, [<Xs>]!, <Xn>!
```

Main variant

Applies when `op1 == 01`.

```
CPYMN [<Xd>]!, [<Xs>]!, <Xn>!
```

Prologue variant

Applies when `op1 == 00`.

```
CPYPN [<Xd>]!, [<Xs>]!, <Xn>!
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_MOPS) || sz != '00' then UNDEFINED;

CPYParams memcpy;
memcpy.d = UInt(Rd);
memcpy.s = UInt(Rs);
memcpy.n = UInt(Rn);
bits(4) options = op2;
```

```
boolean rnontemporal = options<3> == '1';
boolean wnontemporal = options<2> == '1';

case op1 of
    when '00' memcpy.stage = MOPSStage_Prologue;
    when '01' memcpy.stage = MOPSStage_Main;
    when '10' memcpy.stage = MOPSStage_Epilogue;
    otherwise SEE "Memory Copy and Memory Set";

CheckMOPSEnabled();

if (memcpy.s == memcpy.n || memcpy.s == memcpy.d || memcpy.n == memcpy.d || memcpy.d == 31 || memcpy.s
== 31 || memcpy.n == 31) then
    Constraint c = ConstrainUnpredictable(Unpredictable_MOPSOVERLAP31);
    assert c IN {Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNDEF UNDEFINED;
        when Constraint_NOP   EndOfInstruction();
```

Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*, and particularly *Memory Copy and Memory Set CPY**.

Assembler symbols

<Xd> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the "Rd" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.

<Xs> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the "Rs" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the "Rs" field.

<Xn> For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero at the end of the instruction, encoded in the "Rn" field.

For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the "Rn" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the "Rn" field.

Operation

```
constant integer N = MaxBlockSizeCopiedBytes();
bits(8*N) readdata;

memcpy.nzcv = PSTATE.<N,Z,C,V>;
memcpy.toaddress = X[memcpy.d, 64];
memcpy.fromaddress = X[memcpy.s, 64];

if memcpy.stage == MOPSStage_Prologue then
    memcpy.cpysize = UInt(X[memcpy.n, 64]);
else
    memcpy.cpysize = SInt(X[memcpy.n, 64]);

memcpy.implements_option_a = CPYOptionA();
```

```
boolean rprivileged = if options<1> == '1' then AArch64.IsUnprivAccessPriv() else PSTATE.EL != EL0;
boolean wprivileged = if options<0> == '1' then AArch64.IsUnprivAccessPriv() else PSTATE.EL != EL0;

AccessDescriptor raccdesc = CreateAccDescMOPS(MemOp_LOAD, rprivileged, rnontemporal);
AccessDescriptor waccdesc = CreateAccDescMOPS(MemOp_STORE, wprivileged, wnontemporal);

if memcpy.stage == MOPSStage_Prologue then
    if memcpy.cpysize > 0x007FFFFFFFFFFFF then
        memcpy.cpysize = 0x007FFFFFFFFFFFF;

    memcpy.forward = IsMemCpyForward(memcpy);

    if memcpy.implements_option_a then
        memcpy.nzcv = '0000';
        if memcpy.forward then
            // Copy in the forward direction offsets the arguments.
            memcpy.toaddress = memcpy.toaddress + memcpy.cpysize;
            memcpy.fromaddress = memcpy.fromaddress + memcpy.cpysize;
            memcpy.cpysize = 0 - memcpy.cpysize;
    else
        if !memcpy.forward then
            // Copy in the reverse direction offsets the arguments.
            memcpy.toaddress = memcpy.toaddress + memcpy.cpysize;
            memcpy.fromaddress = memcpy.fromaddress + memcpy.cpysize;
            memcpy.nzcv = '1010';
        else
            memcpy.nzcv = '0010';

memcpy.stagecpysize = MemCpyStageSize(memcpy);

if memcpy.stage != MOPSStage_Prologue then
    memcpy.forward = memcpy.cpysize < 0 || (!memcpy.implements_option_a && memcpy.nzcv<3> == '0');
    CheckMemCpyParams(memcpy, options);

integer copied;
boolean iswrite;
AddressDescriptor memaddrdesc;
PhysMemRetStatus memstatus;
boolean fault = FALSE;
integer B;

if memcpy.implements_option_a then
    while memcpy.stagecpysize != 0 && !fault do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(memcpy);

        if memcpy.forward then
            assert B <= -1 * memcpy.stagecpysize;
            (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress + memcpy.cpysize,
memcpy.fromaddress + memcpy.cpysize, memcpy.forward, B, raccdesc, waccdesc);
            if copied != B then
                fault = TRUE;
            else
                memcpy.cpysize = memcpy.cpysize + B;
                memcpy.stagecpysize = memcpy.stagecpysize + B;

        else
            assert B <= memcpy.stagecpysize;
            memcpy.cpysize = memcpy.cpysize - B;
            memcpy.stagecpysize = memcpy.stagecpysize - B;

            (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress + memcpy.cpysize,
memcpy.fromaddress + memcpy.cpysize, memcpy.forward, B, raccdesc, waccdesc);
            if copied != B then
                fault = TRUE;
                memcpy.cpysize = memcpy.cpysize + B;
                memcpy.stagecpysize = memcpy.stagecpysize + B;
```

```
else
    while memcpy.stagecpysize > 0 && !fault do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(memcpy);
        assert B <= memcpy.stagecpysize;

        if memcpy.forward then
            (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress,
memcpy.fromaddress, memcpy.forward, B, raccdesc, waccdesc);
            if copied != B then
                fault = TRUE;
            else
                memcpy.fromaddress = memcpy.fromaddress + B;
                memcpy.toaddress = memcpy.toaddress + B;
        else
            (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress - B,
memcpy.fromaddress - B, memcpy.forward, B, raccdesc, waccdesc);

            if copied != B then
                fault = TRUE;
            else
                memcpy.fromaddress = memcpy.fromaddress - B;
                memcpy.toaddress = memcpy.toaddress - B;

        if !fault then
            memcpy.cpysize = memcpy.cpysize - B;
            memcpy.stagecpysize = memcpy.stagecpysize - B;

UpdateCpyRegisters(memcpy, fault, copied);

if fault then
    if IsFault(memaddrdesc) then
        AArch64.Abort(memaddrdesc.vaddress, memaddrdesc.fault);
    else
        AccessDescriptor accdesc = if iswrite then waccdesc else raccdesc;
        HandleExternalAbort(memstatus, iswrite, memaddrdesc, B, accdesc);

if memcpy.stage == MOPSStage_Prologue then
    PSTATE.<N,Z,C,V> = memcpy.nzcv;
```

C6.2.91 CPYPRN, CPYMRN, CPYERN

Memory Copy, reads non-temporal. These instructions perform a memory copy. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYPRN, then CPYMRN, and then CPYERN.

CPYPRN performs some preconditioning of the arguments suitable for using the CPYMRN instruction, and performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYMRN performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYERN performs the last part of the memory copy.

Note

The inclusion of IMPLEMENTATION DEFINED amounts of memory copy allows some optimization of the size that can be performed.

For CPYPRN, the following saturation logic is applied:

If Xn<63:55> != 000000000, the copy size Xn is saturated to `0x007FFFFFFFFFFFFF`.

After that saturation logic is applied, the direction of the memory copy is based on the following algorithm:

If (Xs > Xd) && (Xd + saturated Xn) > Xs, then direction = forward

Elsif (Xs < Xd) && (Xs + saturated Xn) > Xd, then direction = backward

Else direction = IMPLEMENTATION DEFINED choice between forward and backward.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

After execution of CPYPRN, option A (which results in encoding PSTATE.C = 0):

- PSTATE.{N,Z,V} are set to {0,0,0}.
- If the copy is in the forward direction, then:
  - Xs holds the original Xs + saturated Xn.
  - Xd holds the original Xd + saturated Xn.
  - Xn holds -1* saturated Xn + an IMPLEMENTATION DEFINED number of bytes copied.
- If the copy is in the backward direction, then:
  - Xs and Xd are unchanged.
  - Xn holds the saturated value of Xn - an IMPLEMENTATION DEFINED number of bytes copied.

After execution of CPYPRN, option B (which results in encoding PSTATE.C = 1):

- If the copy is in the forward direction, then:
  - Xs holds the original Xs + an IMPLEMENTATION DEFINED number of bytes copied.
  - Xd holds the original Xd + an IMPLEMENTATION DEFINED number of bytes copied.
  - Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - PSTATE.{N,Z,V} are set to {0,0,0}.
- If the copy is in the backward direction, then:
  - Xs holds the original Xs + saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - Xd holds the original Xd + saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - PSTATE.{N,Z,V} are set to {1,0,0}.

For CPYMRN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number.
- If the copy is in the forward direction (Xn is a negative number), then:
  - Xn holds -1* the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the lowest address that the copy is copied from -Xn.
  - Xd holds the lowest address that the copy is made to -Xn.
  - At the end of the instruction, the value of Xn is written back with -1* the number of bytes remaining to be copied in the memory copy in total.
- If the copy is in the backward direction (Xn is a positive number), then:
  - Xn holds the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the highest address that the copy is copied from -Xn+1.
  - Xd holds the highest address that the copy is copied to -Xn+1.
  - At the end of the instruction, the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.

For CPYMRN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes to be copied in the memory copy in total.
- If the copy is in the forward direction (PSTATE.N == 0), then:
  - Xs holds the lowest address that the copy is copied from.
  - Xd holds the lowest address that the copy is copied to.
  - At the end of the instruction:
    - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
    - the value of Xs is written back with the lowest address that has not been copied from.
    - the value of Xd is written back with the lowest address that has not been copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
  - Xs holds the highest address that the copy is copied from +1.
  - Xd holds the highest address that the copy is copied to +1.
  - At the end of the instruction:
    - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
    - the value of Xs is written back with the highest address that has not been copied from +1.
    - the value of Xd is written back with the highest address that has not been copied to +1.

For CPYERN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number.
- If the copy is in the forward direction (Xn is a negative number), then:
  - Xn holds -1* the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the lowest address that the copy is copied from -Xn.
  - Xd holds the lowest address that the copy is made to -Xn.
  - At the end of the instruction, the value of Xn is written back with 0.
- If the copy is in the backward direction (Xn is a positive number), then:
  - Xn holds the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the highest address that the copy is copied from -Xn+1.

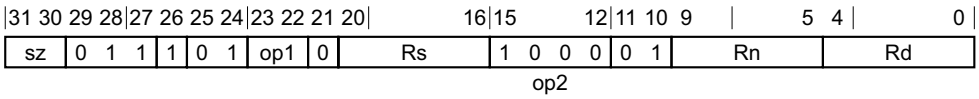
— Xd holds the highest address that the copy is copied to -Xn+1.
— At the end of the instruction, the value of Xn is written back with 0.

For CPYERN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes to be copied in the memory copy in total.
- If the copy is in the forward direction (PSTATE.N == 0), then:
  — Xs holds the lowest address that the copy is copied from.
  — Xd holds the lowest address that the copy is copied to.
  — At the end of the instruction:
    — the value of Xn is written back with 0.
    — the value of Xs is written back with the lowest address that has not been copied from.
    — the value of Xd is written back with the lowest address that has not been copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
  — Xs holds the highest address that the copy is copied from +1.
  — Xd holds the highest address that the copy is copied to +1.
  — At the end of the instruction:
    — the value of Xn is written back with 0.
    — the value of Xs is written back with the highest address that has not been copied from +1.
    — the value of Xd is written back with the highest address that has not been copied to +1.

Integer

(FEAT_MOPS)



Epilogue variant

Applies when `op1 == 10`.

```
CPYERN [<Xd>]!, [<Xs>]!, <Xn>!
```

Main variant

Applies when `op1 == 01`.

```
CPYMRN [<Xd>]!, [<Xs>]!, <Xn>!
```

Prologue variant

Applies when `op1 == 00`.

```
CPYPRN [<Xd>]!, [<Xs>]!, <Xn>!
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_MOPS) || sz != '00' then UNDEFINED;

CPYParams memcpy;
memcpy.d = UInt(Rd);
memcpy.s = UInt(Rs);
memcpy.n = UInt(Rn);
bits(4) options = op2;
```

```
boolean rnontemporal = options<3> == '1';
boolean wnontemporal = options<2> == '1';

case op1 of
    when '00' memcpy.stage = MOPSStage_Prologue;
    when '01' memcpy.stage = MOPSStage_Main;
    when '10' memcpy.stage = MOPSStage_Epilogue;
    otherwise SEE "Memory Copy and Memory Set";

CheckMOPSEnabled();

if (memcpy.s == memcpy.n || memcpy.s == memcpy.d || memcpy.n == memcpy.d || memcpy.d == 31 || memcpy.s
== 31 || memcpy.n == 31) then
    Constraint c = ConstrainUnpredictable(Unpredictable_MOPSOVERLAP31);
    assert c IN {Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNDEF UNDEFINED;
        when Constraint_NOP   EndOfInstruction();
```

Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*, and particularly *Memory Copy and Memory Set CPY**.

Assembler symbols

<Xd> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the "Rd" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.

<Xs> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the "Rs" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the "Rs" field.

<Xn> For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero at the end of the instruction, encoded in the "Rn" field.

For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the "Rn" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the "Rn" field.

Operation

```
constant integer N = MaxBlockSizeCopiedBytes();
bits(8*N) readdata;

memcpy.nzcv = PSTATE.<N,Z,C,V>;
memcpy.toaddress = X[memcpy.d, 64];
memcpy.fromaddress = X[memcpy.s, 64];

if memcpy.stage == MOPSStage_Prologue then
    memcpy.cpysize = UInt(X[memcpy.n, 64]);
else
    memcpy.cpysize = SInt(X[memcpy.n, 64]);

memcpy.implements_option_a = CPYOptionA();
```

```
boolean rprivileged = if options<1> == '1' then AArch64.IsUnprivAccessPriv() else PSTATE.EL != EL0;
boolean wprivileged = if options<0> == '1' then AArch64.IsUnprivAccessPriv() else PSTATE.EL != EL0;

AccessDescriptor raccdesc = CreateAccDescMOPS(MemOp_LOAD, rprivileged, rnontemporal);
AccessDescriptor waccdesc = CreateAccDescMOPS(MemOp_STORE, wprivileged, wnontemporal);

if memcpy.stage == MOPSStage_Prologue then
    if memcpy.cpysize > 0x007FFFFFFFFFFFF then
        memcpy.cpysize = 0x007FFFFFFFFFFFF;

    memcpy.forward = IsMemCpyForward(memcpy);

    if memcpy.implements_option_a then
        memcpy.nzcv = '0000';
        if memcpy.forward then
            // Copy in the forward direction offsets the arguments.
            memcpy.toaddress = memcpy.toaddress + memcpy.cpysize;
            memcpy.fromaddress = memcpy.fromaddress + memcpy.cpysize;
            memcpy.cpysize = 0 - memcpy.cpysize;
    else
        if !memcpy.forward then
            // Copy in the reverse direction offsets the arguments.
            memcpy.toaddress = memcpy.toaddress + memcpy.cpysize;
            memcpy.fromaddress = memcpy.fromaddress + memcpy.cpysize;
            memcpy.nzcv = '1010';
        else
            memcpy.nzcv = '0010';

memcpy.stagecpysize = MemCpyStageSize(memcpy);

if memcpy.stage != MOPSStage_Prologue then
    memcpy.forward = memcpy.cpysize < 0 || (!memcpy.implements_option_a && memcpy.nzcv<3> == '0');
    CheckMemCpyParams(memcpy, options);

integer copied;
boolean iswrite;
AddressDescriptor memaddrdesc;
PhysMemRetStatus memstatus;
boolean fault = FALSE;
integer B;

if memcpy.implements_option_a then
    while memcpy.stagecpysize != 0 && !fault do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(memcpy);

        if memcpy.forward then
            assert B <= -1 * memcpy.stagecpysize;
            (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress + memcpy.cpysize,
memcpy.fromaddress + memcpy.cpysize, memcpy.forward, B, raccdesc, waccdesc);
            if copied != B then
                fault = TRUE;
            else
                memcpy.cpysize = memcpy.cpysize + B;
                memcpy.stagecpysize = memcpy.stagecpysize + B;

        else
            assert B <= memcpy.stagecpysize;
            memcpy.cpysize = memcpy.cpysize - B;
            memcpy.stagecpysize = memcpy.stagecpysize - B;

            (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress + memcpy.cpysize,
memcpy.fromaddress + memcpy.cpysize, memcpy.forward, B, raccdesc, waccdesc);
            if copied != B then
                fault = TRUE;
                memcpy.cpysize = memcpy.cpysize + B;
                memcpy.stagecpysize = memcpy.stagecpysize + B;
```

```
else
    while memcpy.stagecpysize > 0 && !fault do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(memcpy);
        assert B <= memcpy.stagecpysize;

        if memcpy.forward then
            (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress,
memcpy.fromaddress, memcpy.forward, B, raccdesc, waccdesc);
            if copied != B then
                fault = TRUE;
            else
                memcpy.fromaddress = memcpy.fromaddress + B;
                memcpy.toaddress = memcpy.toaddress + B;
        else
            (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress - B,
memcpy.fromaddress - B, memcpy.forward, B, raccdesc, waccdesc);

            if copied != B then
                fault = TRUE;
            else
                memcpy.fromaddress = memcpy.fromaddress - B;
                memcpy.toaddress = memcpy.toaddress - B;

        if !fault then
            memcpy.cpysize = memcpy.cpysize - B;
            memcpy.stagecpysize = memcpy.stagecpysize - B;

UpdateCpyRegisters(memcpy, fault, copied);

if fault then
    if IsFault(memaddrdesc) then
        AArch64.Abort(memaddrdesc.vaddress, memaddrdesc.fault);
    else
        AccessDescriptor accdesc = if iswrite then waccdesc else raccdesc;
        HandleExternalAbort(memstatus, iswrite, memaddrdesc, B, accdesc);

if memcpy.stage == MOPSStage_Prologue then
    PSTATE.<N,Z,C,V> = memcpy.nzcv;
```

C6.2.92 CPYPRT, CPYMRT, CPYERT

Memory Copy, reads unprivileged. These instructions perform a memory copy. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYPRT, then CPYMRT, and then CPYERT.

CPYPRT performs some preconditioning of the arguments suitable for using the CPYMRT instruction, and performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYMRT performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYERT performs the last part of the memory copy.

Note

The inclusion of IMPLEMENTATION DEFINED amounts of memory copy allows some optimization of the size that can be performed.

For CPYPRT, the following saturation logic is applied:

If Xn<63:55> != 000000000, the copy size Xn is saturated to `0x007FFFFFFFFFFFF`.

After that saturation logic is applied, the direction of the memory copy is based on the following algorithm:

If (Xs > Xd) && (Xd + saturated Xn) > Xs, then direction = forward

Elsif (Xs < Xd) && (Xs + saturated Xn) > Xd, then direction = backward

Else direction = IMPLEMENTATION DEFINED choice between forward and backward.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

After execution of CPYPRT, option A (which results in encoding PSTATE.C = 0):

- PSTATE.{N,Z,V} are set to {0,0,0}.
- If the copy is in the forward direction, then:
  - Xs holds the original Xs + saturated Xn.
  - Xd holds the original Xd + saturated Xn.
  - Xn holds -1* saturated Xn + an IMPLEMENTATION DEFINED number of bytes copied.
- If the copy is in the backward direction, then:
  - Xs and Xd are unchanged.
  - Xn holds the saturated value of Xn - an IMPLEMENTATION DEFINED number of bytes copied.

After execution of CPYPRT, option B (which results in encoding PSTATE.C = 1):

- If the copy is in the forward direction, then:
  - Xs holds the original Xs + an IMPLEMENTATION DEFINED number of bytes copied.
  - Xd holds the original Xd + an IMPLEMENTATION DEFINED number of bytes copied.
  - Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - PSTATE.{N,Z,V} are set to {0,0,0}.
- If the copy is in the backward direction, then:
  - Xs holds the original Xs + saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - Xd holds the original Xd + saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - PSTATE.{N,Z,V} are set to {1,0,0}.

For CPYMRT, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number.
- If the copy is in the forward direction (Xn is a negative number), then:
  - Xn holds -1* the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the lowest address that the copy is copied from -Xn.
  - Xd holds the lowest address that the copy is made to -Xn.
  - At the end of the instruction, the value of Xn is written back with -1* the number of bytes remaining to be copied in the memory copy in total.
- If the copy is in the backward direction (Xn is a positive number), then:
  - Xn holds the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the highest address that the copy is copied from -Xn+1.
  - Xd holds the highest address that the copy is copied to -Xn+1.
  - At the end of the instruction, the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.

For CPYMRT, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes to be copied in the memory copy in total.
- If the copy is in the forward direction (PSTATE.N == 0), then:
  - Xs holds the lowest address that the copy is copied from.
  - Xd holds the lowest address that the copy is copied to.
  - At the end of the instruction:
    - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
    - the value of Xs is written back with the lowest address that has not been copied from.
    - the value of Xd is written back with the lowest address that has not been copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
  - Xs holds the highest address that the copy is copied from +1.
  - Xd holds the highest address that the copy is copied to +1.
  - At the end of the instruction:
    - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
    - the value of Xs is written back with the highest address that has not been copied from +1.
    - the value of Xd is written back with the highest address that has not been copied to +1.

For CPYERT, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number.
- If the copy is in the forward direction (Xn is a negative number), then:
  - Xn holds -1* the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the lowest address that the copy is copied from -Xn.
  - Xd holds the lowest address that the copy is made to -Xn.
  - At the end of the instruction, the value of Xn is written back with 0.
- If the copy is in the backward direction (Xn is a positive number), then:
  - Xn holds the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the highest address that the copy is copied from -Xn+1.

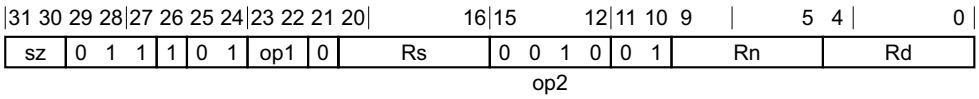
- Xd holds the highest address that the copy is copied to -Xn+1.
- At the end of the instruction, the value of Xn is written back with 0.

For CPYERT, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes to be copied in the memory copy in total.
- If the copy is in the forward direction (PSTATE.N == 0), then:
  - Xs holds the lowest address that the copy is copied from.
  - Xd holds the lowest address that the copy is copied to.
  - At the end of the instruction:
    - the value of Xn is written back with 0.
    - the value of Xs is written back with the lowest address that has not been copied from.
    - the value of Xd is written back with the lowest address that has not been copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
  - Xs holds the highest address that the copy is copied from +1.
  - Xd holds the highest address that the copy is copied to +1.
  - At the end of the instruction:
    - the value of Xn is written back with 0.
    - the value of Xs is written back with the highest address that has not been copied from +1.
    - the value of Xd is written back with the highest address that has not been copied to +1.

Integer

(FEAT_MOPS)



Epilogue variant

Applies when `op1 == 10`.

```
CPYERT [<Xd>]!, [<Xs>]!, <Xn>!
```

Main variant

Applies when `op1 == 01`.

```
CPYMRT [<Xd>]!, [<Xs>]!, <Xn>!
```

Prologue variant

Applies when `op1 == 00`.

```
CPYPRT [<Xd>]!, [<Xs>]!, <Xn>!
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_MOPS) || sz != '00' then UNDEFINED;

CPYParams memcpy;
memcpy.d = UInt(Rd);
memcpy.s = UInt(Rs);
memcpy.n = UInt(Rn);
bits(4) options = op2;
```

```
boolean rnontemporal = options<3> == '1';
boolean wnontemporal = options<2> == '1';

case op1 of
    when '00' memcpy.stage = MOPSStage_Prologue;
    when '01' memcpy.stage = MOPSStage_Main;
    when '10' memcpy.stage = MOPSStage_Epilogue;
    otherwise SEE "Memory Copy and Memory Set";

CheckMOPSEnabled();

if (memcpy.s == memcpy.n || memcpy.s == memcpy.d || memcpy.n == memcpy.d || memcpy.d == 31 || memcpy.s
== 31 || memcpy.n == 31) then
    Constraint c = ConstrainUnpredictable(Unpredictable_MOPSOVERLAP31);
    assert c IN {Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNDEF UNDEFINED;
        when Constraint_NOP   EndOfInstruction();
```

Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*, and particularly *Memory Copy and Memory Set CPY**.

Assembler symbols

<Xd> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the "Rd" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.

<Xs> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the "Rs" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the "Rs" field.

<Xn> For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero at the end of the instruction, encoded in the "Rn" field.

For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the "Rn" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the "Rn" field.

Operation

```
constant integer N = MaxBlockSizeCopiedBytes();
bits(8*N) readdata;

memcpy.nzcv = PSTATE.<N,Z,C,V>;
memcpy.toaddress = X[memcpy.d, 64];
memcpy.fromaddress = X[memcpy.s, 64];

if memcpy.stage == MOPSStage_Prologue then
    memcpy.cpysize = UInt(X[memcpy.n, 64]);
else
    memcpy.cpysize = SInt(X[memcpy.n, 64]);

memcpy.implements_option_a = CPYOptionA();
```

```
boolean rprivileged = if options<1> == '1' then AArch64.IsUnprivAccessPriv() else PSTATE.EL != EL0;
boolean wprivileged = if options<0> == '1' then AArch64.IsUnprivAccessPriv() else PSTATE.EL != EL0;

AccessDescriptor raccdesc = CreateAccDescMOPS(MemOp_LOAD, rprivileged, rnontemporal);
AccessDescriptor waccdesc = CreateAccDescMOPS(MemOp_STORE, wprivileged, wnontemporal);

if memcpy.stage == MOPSStage_Prologue then
    if memcpy.cpysize > 0x007FFFFFFFFFFFF then
        memcpy.cpysize = 0x007FFFFFFFFFFFF;

    memcpy.forward = IsMemCpyForward(memcpy);

    if memcpy.implements_option_a then
        memcpy.nzcv = '0000';
        if memcpy.forward then
            // Copy in the forward direction offsets the arguments.
            memcpy.toaddress = memcpy.toaddress + memcpy.cpysize;
            memcpy.fromaddress = memcpy.fromaddress + memcpy.cpysize;
            memcpy.cpysize = 0 - memcpy.cpysize;
    else
        if !memcpy.forward then
            // Copy in the reverse direction offsets the arguments.
            memcpy.toaddress = memcpy.toaddress + memcpy.cpysize;
            memcpy.fromaddress = memcpy.fromaddress + memcpy.cpysize;
            memcpy.nzcv = '1010';
        else
            memcpy.nzcv = '0010';

memcpy.stagecpysize = MemCpyStageSize(memcpy);

if memcpy.stage != MOPSStage_Prologue then
    memcpy.forward = memcpy.cpysize < 0 || (!memcpy.implements_option_a && memcpy.nzcv<3> == '0');
    CheckMemCpyParams(memcpy, options);

integer copied;
boolean iswrite;
AddressDescriptor memaddrdesc;
PhysMemRetStatus memstatus;
boolean fault = FALSE;
integer B;

if memcpy.implements_option_a then
    while memcpy.stagecpysize != 0 && !fault do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(memcpy);

        if memcpy.forward then
            assert B <= -1 * memcpy.stagecpysize;
            (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress + memcpy.cpysize,
memcpy.fromaddress + memcpy.cpysize, memcpy.forward, B, raccdesc, waccdesc);
            if copied != B then
                fault = TRUE;
            else
                memcpy.cpysize = memcpy.cpysize + B;
                memcpy.stagecpysize = memcpy.stagecpysize + B;

        else
            assert B <= memcpy.stagecpysize;
            memcpy.cpysize = memcpy.cpysize - B;
            memcpy.stagecpysize = memcpy.stagecpysize - B;

            (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress + memcpy.cpysize,
memcpy.fromaddress + memcpy.cpysize, memcpy.forward, B, raccdesc, waccdesc);
            if copied != B then
                fault = TRUE;
                memcpy.cpysize = memcpy.cpysize + B;
                memcpy.stagecpysize = memcpy.stagecpysize + B;
```

```
else
    while memcpy.stagecpysize > 0 && !fault do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(memcpy);
        assert B <= memcpy.stagecpysize;

        if memcpy.forward then
            (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress,
memcpy.fromaddress, memcpy.forward, B, raccdesc, waccdesc);
            if copied != B then
                fault = TRUE;
            else
                memcpy.fromaddress = memcpy.fromaddress + B;
                memcpy.toaddress = memcpy.toaddress + B;
        else
            (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress - B,
memcpy.fromaddress - B, memcpy.forward, B, raccdesc, waccdesc);

            if copied != B then
                fault = TRUE;
            else
                memcpy.fromaddress = memcpy.fromaddress - B;
                memcpy.toaddress = memcpy.toaddress - B;

        if !fault then
            memcpy.cpysize = memcpy.cpysize - B;
            memcpy.stagecpysize = memcpy.stagecpysize - B;

UpdateCpyRegisters(memcpy, fault, copied);

if fault then
    if IsFault(memaddrdesc) then
        AArch64.Abort(memaddrdesc.vaddress, memaddrdesc.fault);
    else
        AccessDescriptor accdesc = if iswrite then waccdesc else raccdesc;
        HandleExternalAbort(memstatus, iswrite, memaddrdesc, B, accdesc);

if memcpy.stage == MOPSStage_Prologue then
    PSTATE.<N,Z,C,V> = memcpy.nzcv;
```

C6.2.93 CPYPRTN, CPYMRTN, CPYERTN

Memory Copy, reads unprivileged, reads and writes non-temporal. These instructions perform a memory copy. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYPRTN, then CPYMRTN, and then CPYERTN.

CPYPRTN performs some preconditioning of the arguments suitable for using the CPYMRTN instruction, and performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYMRTN performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYERTN performs the last part of the memory copy.

Note

The inclusion of IMPLEMENTATION DEFINED amounts of memory copy allows some optimization of the size that can be performed.

For CPYPRTN, the following saturation logic is applied:

If Xn<63:55> != 000000000, the copy size Xn is saturated to `0x007FFFFFFFFFFFF`.

After that saturation logic is applied, the direction of the memory copy is based on the following algorithm:

If (Xs > Xd) && (Xd + saturated Xn) > Xs, then direction = forward

Elsif (Xs < Xd) && (Xs + saturated Xn) > Xd, then direction = backward

Else direction = IMPLEMENTATION DEFINED choice between forward and backward.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

After execution of CPYPRTN, option A (which results in encoding PSTATE.C = 0):

- PSTATE.{N,Z,V} are set to {0,0,0}.
- If the copy is in the forward direction, then:
  - Xs holds the original Xs + saturated Xn.
  - Xd holds the original Xd + saturated Xn.
  - Xn holds -1* saturated Xn + an IMPLEMENTATION DEFINED number of bytes copied.
- If the copy is in the backward direction, then:
  - Xs and Xd are unchanged.
  - Xn holds the saturated value of Xn - an IMPLEMENTATION DEFINED number of bytes copied.

After execution of CPYPRTN, option B (which results in encoding PSTATE.C = 1):

- If the copy is in the forward direction, then:
  - Xs holds the original Xs + an IMPLEMENTATION DEFINED number of bytes copied.
  - Xd holds the original Xd + an IMPLEMENTATION DEFINED number of bytes copied.
  - Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - PSTATE.{N,Z,V} are set to {0,0,0}.
- If the copy is in the backward direction, then:
  - Xs holds the original Xs + saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - Xd holds the original Xd + saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - PSTATE.{N,Z,V} are set to {1,0,0}.

For CPYMRTN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number.
- If the copy is in the forward direction (Xn is a negative number), then:
  - Xn holds -1* the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the lowest address that the copy is copied from -Xn.
  - Xd holds the lowest address that the copy is made to -Xn.
  - At the end of the instruction, the value of Xn is written back with -1* the number of bytes remaining to be copied in the memory copy in total.
- If the copy is in the backward direction (Xn is a positive number), then:
  - Xn holds the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the highest address that the copy is copied from -Xn+1.
  - Xd holds the highest address that the copy is copied to -Xn+1.
  - At the end of the instruction, the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.

For CPYMRTN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes to be copied in the memory copy in total.
- If the copy is in the forward direction (PSTATE.N == 0), then:
  - Xs holds the lowest address that the copy is copied from.
  - Xd holds the lowest address that the copy is copied to.
  - At the end of the instruction:
    - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
    - the value of Xs is written back with the lowest address that has not been copied from.
    - the value of Xd is written back with the lowest address that has not been copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
  - Xs holds the highest address that the copy is copied from +1.
  - Xd holds the highest address that the copy is copied to +1.
  - At the end of the instruction:
    - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
    - the value of Xs is written back with the highest address that has not been copied from +1.
    - the value of Xd is written back with the highest address that has not been copied to +1.

For CPYERTN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number.
- If the copy is in the forward direction (Xn is a negative number), then:
  - Xn holds -1* the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the lowest address that the copy is copied from -Xn.
  - Xd holds the lowest address that the copy is made to -Xn.
  - At the end of the instruction, the value of Xn is written back with 0.
- If the copy is in the backward direction (Xn is a positive number), then:
  - Xn holds the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the highest address that the copy is copied from -Xn+1.

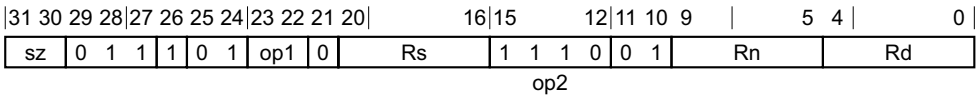
  - Xd holds the highest address that the copy is copied to -Xn+1.
  - At the end of the instruction, the value of Xn is written back with 0.

For CPYERTN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes to be copied in the memory copy in total.
- If the copy is in the forward direction (PSTATE.N == 0), then:
  - Xs holds the lowest address that the copy is copied from.
  - Xd holds the lowest address that the copy is copied to.
  - At the end of the instruction:
    - the value of Xn is written back with 0.
    - the value of Xs is written back with the lowest address that has not been copied from.
    - the value of Xd is written back with the lowest address that has not been copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
  - Xs holds the highest address that the copy is copied from +1.
  - Xd holds the highest address that the copy is copied to +1.
  - At the end of the instruction:
    - the value of Xn is written back with 0.
    - the value of Xs is written back with the highest address that has not been copied from +1.
    - the value of Xd is written back with the highest address that has not been copied to +1.

Integer

(FEAT_MOPS)



Epilogue variant

Applies when `op1 == 10`.

```
CPYERTN [<Xd>]!, [<Xs>]!, <Xn>!
```

Main variant

Applies when `op1 == 01`.

```
CPYMRTN [<Xd>]!, [<Xs>]!, <Xn>!
```

Prologue variant

Applies when `op1 == 00`.

```
CPYPRTN [<Xd>]!, [<Xs>]!, <Xn>!
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_MOPS) || sz != '00' then UNDEFINED;

CPYParams memcpy;
memcpy.d = UInt(Rd);
memcpy.s = UInt(Rs);
memcpy.n = UInt(Rn);
bits(4) options = op2;
```

```
boolean rnontemporal = options<3> == '1';
boolean wnontemporal = options<2> == '1';

case op1 of
    when '00' memcpy.stage = MOPSStage_Prologue;
    when '01' memcpy.stage = MOPSStage_Main;
    when '10' memcpy.stage = MOPSStage_Epilogue;
    otherwise SEE "Memory Copy and Memory Set";

CheckMOPSEnabled();

if (memcpy.s == memcpy.n || memcpy.s == memcpy.d || memcpy.n == memcpy.d || memcpy.d == 31 || memcpy.s
== 31 || memcpy.n == 31) then
    Constraint c = ConstrainUnpredictable(Unpredictable_MOPSOVERLAP31);
    assert c IN {Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNDEF UNDEFINED;
        when Constraint_NOP   EndOfInstruction();
```

Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*, and particularly *Memory Copy and Memory Set CPY**.

Assembler symbols

<Xd> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the "Rd" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.

<Xs> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the "Rs" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the "Rs" field.

<Xn> For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero at the end of the instruction, encoded in the "Rn" field.

For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the "Rn" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the "Rn" field.

Operation

```
constant integer N = MaxBlockSizeCopiedBytes();
bits(8*N) readdata;

memcpy.nzcv = PSTATE.<N,Z,C,V>;
memcpy.toaddress = X[memcpy.d, 64];
memcpy.fromaddress = X[memcpy.s, 64];

if memcpy.stage == MOPSStage_Prologue then
    memcpy.cpysize = UInt(X[memcpy.n, 64]);
else
    memcpy.cpysize = SInt(X[memcpy.n, 64]);

memcpy.implements_option_a = CPYOptionA();
```

```
boolean rprivileged = if options<1> == '1' then AArch64.IsUnprivAccessPriv() else PSTATE.EL != EL0;
boolean wprivileged = if options<0> == '1' then AArch64.IsUnprivAccessPriv() else PSTATE.EL != EL0;

AccessDescriptor raccdesc = CreateAccDescMOPS(MemOp_LOAD, rprivileged, rnontemporal);
AccessDescriptor waccdesc = CreateAccDescMOPS(MemOp_STORE, wprivileged, wnontemporal);

if memcpy.stage == MOPSStage_Prologue then
    if memcpy.cpysize > 0x007FFFFFFFFFFFF then
        memcpy.cpysize = 0x007FFFFFFFFFFFF;

    memcpy.forward = IsMemCpyForward(memcpy);

    if memcpy.implements_option_a then
        memcpy.nzcv = '0000';
        if memcpy.forward then
            // Copy in the forward direction offsets the arguments.
            memcpy.toaddress = memcpy.toaddress + memcpy.cpysize;
            memcpy.fromaddress = memcpy.fromaddress + memcpy.cpysize;
            memcpy.cpysize = 0 - memcpy.cpysize;
    else
        if !memcpy.forward then
            // Copy in the reverse direction offsets the arguments.
            memcpy.toaddress = memcpy.toaddress + memcpy.cpysize;
            memcpy.fromaddress = memcpy.fromaddress + memcpy.cpysize;
            memcpy.nzcv = '1010';
        else
            memcpy.nzcv = '0010';

memcpy.stagecpysize = MemCpyStageSize(memcpy);

if memcpy.stage != MOPSStage_Prologue then
    memcpy.forward = memcpy.cpysize < 0 || (!memcpy.implements_option_a && memcpy.nzcv<3> == '0');
    CheckMemCpyParams(memcpy, options);

integer copied;
boolean iswrite;
AddressDescriptor memaddrdesc;
PhysMemRetStatus memstatus;
boolean fault = FALSE;
integer B;

if memcpy.implements_option_a then
    while memcpy.stagecpysize != 0 && !fault do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(memcpy);

        if memcpy.forward then
            assert B <= -1 * memcpy.stagecpysize;
            (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress + memcpy.cpysize,
memcpy.fromaddress + memcpy.cpysize, memcpy.forward, B, raccdesc, waccdesc);
            if copied != B then
                fault = TRUE;
            else
                memcpy.cpysize = memcpy.cpysize + B;
                memcpy.stagecpysize = memcpy.stagecpysize + B;

        else
            assert B <= memcpy.stagecpysize;
            memcpy.cpysize = memcpy.cpysize - B;
            memcpy.stagecpysize = memcpy.stagecpysize - B;

            (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress + memcpy.cpysize,
memcpy.fromaddress + memcpy.cpysize, memcpy.forward, B, raccdesc, waccdesc);
            if copied != B then
                fault = TRUE;
                memcpy.cpysize = memcpy.cpysize + B;
                memcpy.stagecpysize = memcpy.stagecpysize + B;
```

```
else
    while memcpy.stagecpysize > 0 && !fault do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(memcpy);
        assert B <= memcpy.stagecpysize;

        if memcpy.forward then
            (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress,
memcpy.fromaddress, memcpy.forward, B, raccdesc, waccdesc);
            if copied != B then
                fault = TRUE;
            else
                memcpy.fromaddress = memcpy.fromaddress + B;
                memcpy.toaddress = memcpy.toaddress + B;
        else
            (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress - B,
memcpy.fromaddress - B, memcpy.forward, B, raccdesc, waccdesc);

            if copied != B then
                fault = TRUE;
            else
                memcpy.fromaddress = memcpy.fromaddress - B;
                memcpy.toaddress = memcpy.toaddress - B;

        if !fault then
            memcpy.cpysize = memcpy.cpysize - B;
            memcpy.stagecpysize = memcpy.stagecpysize - B;

UpdateCpyRegisters(memcpy, fault, copied);

if fault then
    if IsFault(memaddrdesc) then
        AArch64.Abort(memaddrdesc.vaddress, memaddrdesc.fault);
    else
        AccessDescriptor accdesc = if iswrite then waccdesc else raccdesc;
        HandleExternalAbort(memstatus, iswrite, memaddrdesc, B, accdesc);

if memcpy.stage == MOPSStage_Prologue then
    PSTATE.<N,Z,C,V> = memcpy.nzcv;
```

C6.2.94 CPYPRTRN, CPYMRTRN, CPYERTRN

Memory Copy, reads unprivileged and non-temporal. These instructions perform a memory copy. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYPRTRN, then CPYMRTRN, and then CPYERTRN.

CPYPRTRN performs some preconditioning of the arguments suitable for using the CPYMRTRN instruction, and performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYMRTRN performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYERTRN performs the last part of the memory copy.

Note

The inclusion of IMPLEMENTATION DEFINED amounts of memory copy allows some optimization of the size that can be performed.

For CPYPRTRN, the following saturation logic is applied:

If Xn<63:55> != 000000000, the copy size Xn is saturated to `0x007FFFFFFFFFFFFF`.

After that saturation logic is applied, the direction of the memory copy is based on the following algorithm:

If (Xs > Xd) && (Xd + saturated Xn) > Xs, then direction = forward

Elsif (Xs < Xd) && (Xs + saturated Xn) > Xd, then direction = backward

Else direction = IMPLEMENTATION DEFINED choice between forward and backward.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

After execution of CPYPRTRN, option A (which results in encoding PSTATE.C = 0):

- PSTATE.{N,Z,V} are set to {0,0,0}.
- If the copy is in the forward direction, then:
  - Xs holds the original Xs + saturated Xn.
  - Xd holds the original Xd + saturated Xn.
  - Xn holds -1* saturated Xn + an IMPLEMENTATION DEFINED number of bytes copied.
- If the copy is in the backward direction, then:
  - Xs and Xd are unchanged.
  - Xn holds the saturated value of Xn - an IMPLEMENTATION DEFINED number of bytes copied.

After execution of CPYPRTRN, option B (which results in encoding PSTATE.C = 1):

- If the copy is in the forward direction, then:
  - Xs holds the original Xs + an IMPLEMENTATION DEFINED number of bytes copied.
  - Xd holds the original Xd + an IMPLEMENTATION DEFINED number of bytes copied.
  - Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - PSTATE.{N,Z,V} are set to {0,0,0}.
- If the copy is in the backward direction, then:
  - Xs holds the original Xs + saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - Xd holds the original Xd + saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - PSTATE.{N,Z,V} are set to {1,0,0}.

For CPYMRTRN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number.
- If the copy is in the forward direction (Xn is a negative number), then:
  - Xn holds -1* the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the lowest address that the copy is copied from -Xn.
  - Xd holds the lowest address that the copy is made to -Xn.
  - At the end of the instruction, the value of Xn is written back with -1* the number of bytes remaining to be copied in the memory copy in total.
- If the copy is in the backward direction (Xn is a positive number), then:
  - Xn holds the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the highest address that the copy is copied from -Xn+1.
  - Xd holds the highest address that the copy is copied to -Xn+1.
  - At the end of the instruction, the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.

For CPYMRTRN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes to be copied in the memory copy in total.
- If the copy is in the forward direction (PSTATE.N == 0), then:
  - Xs holds the lowest address that the copy is copied from.
  - Xd holds the lowest address that the copy is copied to.
  - At the end of the instruction:
    - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
    - the value of Xs is written back with the lowest address that has not been copied from.
    - the value of Xd is written back with the lowest address that has not been copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
  - Xs holds the highest address that the copy is copied from +1.
  - Xd holds the highest address that the copy is copied to +1.
  - At the end of the instruction:
    - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
    - the value of Xs is written back with the highest address that has not been copied from +1.
    - the value of Xd is written back with the highest address that has not been copied to +1.

For CPYERTRN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number.
- If the copy is in the forward direction (Xn is a negative number), then:
  - Xn holds -1* the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the lowest address that the copy is copied from -Xn.
  - Xd holds the lowest address that the copy is made to -Xn.
  - At the end of the instruction, the value of Xn is written back with 0.
- If the copy is in the backward direction (Xn is a positive number), then:
  - Xn holds the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the highest address that the copy is copied from -Xn+1.

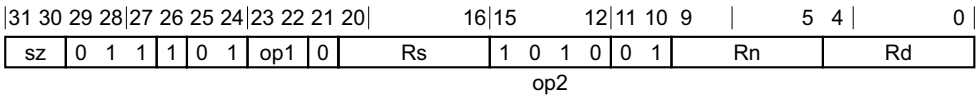
- Xd holds the highest address that the copy is copied to -Xn+1.
- At the end of the instruction, the value of Xn is written back with 0.

For CPYERTRN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes to be copied in the memory copy in total.
- If the copy is in the forward direction (PSTATE.N == 0), then:
  - Xs holds the lowest address that the copy is copied from.
  - Xd holds the lowest address that the copy is copied to.
  - At the end of the instruction:
    - the value of Xn is written back with 0.
    - the value of Xs is written back with the lowest address that has not been copied from.
    - the value of Xd is written back with the lowest address that has not been copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
  - Xs holds the highest address that the copy is copied from +1.
  - Xd holds the highest address that the copy is copied to +1.
  - At the end of the instruction:
    - the value of Xn is written back with 0.
    - the value of Xs is written back with the highest address that has not been copied from +1.
    - the value of Xd is written back with the highest address that has not been copied to +1.

Integer

(FEAT_MOPS)



Epilogue variant

Applies when `op1 == 10`.

```
CPYERTRN [<Xd>]!, [<Xs>]!, <Xn>!
```

Main variant

Applies when `op1 == 01`.

```
CPYMRTRN [<Xd>]!, [<Xs>]!, <Xn>!
```

Prologue variant

Applies when `op1 == 00`.

```
CPYPRTRN [<Xd>]!, [<Xs>]!, <Xn>!
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_MOPS) || sz != '00' then UNDEFINED;

CPYParams memcpy;
memcpy.d = UInt(Rd);
memcpy.s = UInt(Rs);
memcpy.n = UInt(Rn);
bits(4) options = op2;
```

```
boolean rnontemporal = options<3> == '1';
boolean wnontemporal = options<2> == '1';

case op1 of
    when '00' memcpy.stage = MOPSStage_Prologue;
    when '01' memcpy.stage = MOPSStage_Main;
    when '10' memcpy.stage = MOPSStage_Epilogue;
    otherwise SEE "Memory Copy and Memory Set";

CheckMOPSEnabled();

if (memcpy.s == memcpy.n || memcpy.s == memcpy.d || memcpy.n == memcpy.d || memcpy.d == 31 || memcpy.s
== 31 || memcpy.n == 31) then
    Constraint c = ConstrainUnpredictable(Unpredictable_MOPSOVERLAP31);
    assert c IN {Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNDEF UNDEFINED;
        when Constraint_NOP   EndOfInstruction();
```

Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*, and particularly *Memory Copy and Memory Set CPY**.

Assembler symbols

<Xd> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the "Rd" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.

<Xs> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the "Rs" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the "Rs" field.

<Xn> For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero at the end of the instruction, encoded in the "Rn" field.

For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the "Rn" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the "Rn" field.

Operation

```
constant integer N = MaxBlockSizeCopiedBytes();
bits(8*N) readdata;

memcpy.nzcv = PSTATE.<N,Z,C,V>;
memcpy.toaddress = X[memcpy.d, 64];
memcpy.fromaddress = X[memcpy.s, 64];

if memcpy.stage == MOPSStage_Prologue then
    memcpy.cpysize = UInt(X[memcpy.n, 64]);
else
    memcpy.cpysize = SInt(X[memcpy.n, 64]);

memcpy.implements_option_a = CPYOptionA();
```

```
boolean rprivileged = if options<1> == '1' then AArch64.IsUnprivAccessPriv() else PSTATE.EL != EL0;
boolean wprivileged = if options<0> == '1' then AArch64.IsUnprivAccessPriv() else PSTATE.EL != EL0;

AccessDescriptor raccdesc = CreateAccDescMOPS(MemOp_LOAD, rprivileged, rnontemporal);
AccessDescriptor waccdesc = CreateAccDescMOPS(MemOp_STORE, wprivileged, wnontemporal);

if memcpy.stage == MOPSStage_Prologue then
    if memcpy.cpysize > 0x007FFFFFFFFFFFF then
        memcpy.cpysize = 0x007FFFFFFFFFFFF;

    memcpy.forward = IsMemCpyForward(memcpy);

    if memcpy.implements_option_a then
        memcpy.nzcv = '0000';
        if memcpy.forward then
            // Copy in the forward direction offsets the arguments.
            memcpy.toaddress = memcpy.toaddress + memcpy.cpysize;
            memcpy.fromaddress = memcpy.fromaddress + memcpy.cpysize;
            memcpy.cpysize = 0 - memcpy.cpysize;
    else
        if !memcpy.forward then
            // Copy in the reverse direction offsets the arguments.
            memcpy.toaddress = memcpy.toaddress + memcpy.cpysize;
            memcpy.fromaddress = memcpy.fromaddress + memcpy.cpysize;
            memcpy.nzcv = '1010';
        else
            memcpy.nzcv = '0010';

memcpy.stagecpysize = MemCpyStageSize(memcpy);

if memcpy.stage != MOPSStage_Prologue then
    memcpy.forward = memcpy.cpysize < 0 || (!memcpy.implements_option_a && memcpy.nzcv<3> == '0');
    CheckMemCpyParams(memcpy, options);

integer copied;
boolean iswrite;
AddressDescriptor memaddrdesc;
PhysMemRetStatus memstatus;
boolean fault = FALSE;
integer B;

if memcpy.implements_option_a then
    while memcpy.stagecpysize != 0 && !fault do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(memcpy);

        if memcpy.forward then
            assert B <= -1 * memcpy.stagecpysize;
            (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress + memcpy.cpysize,
memcpy.fromaddress + memcpy.cpysize, memcpy.forward, B, raccdesc, waccdesc);
            if copied != B then
                fault = TRUE;
            else
                memcpy.cpysize = memcpy.cpysize + B;
                memcpy.stagecpysize = memcpy.stagecpysize + B;

        else
            assert B <= memcpy.stagecpysize;
            memcpy.cpysize = memcpy.cpysize - B;
            memcpy.stagecpysize = memcpy.stagecpysize - B;

            (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress + memcpy.cpysize,
memcpy.fromaddress + memcpy.cpysize, memcpy.forward, B, raccdesc, waccdesc);
            if copied != B then
                fault = TRUE;
                memcpy.cpysize = memcpy.cpysize + B;
                memcpy.stagecpysize = memcpy.stagecpysize + B;
```

```
else
    while memcpy.stagecpysize > 0 && !fault do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(memcpy);
        assert B <= memcpy.stagecpysize;

        if memcpy.forward then
            (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress,
memcpy.fromaddress, memcpy.forward, B, raccdesc, waccdesc);
            if copied != B then
                fault = TRUE;
            else
                memcpy.fromaddress = memcpy.fromaddress + B;
                memcpy.toaddress = memcpy.toaddress + B;
        else
            (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress - B,
memcpy.fromaddress - B, memcpy.forward, B, raccdesc, waccdesc);

            if copied != B then
                fault = TRUE;
            else
                memcpy.fromaddress = memcpy.fromaddress - B;
                memcpy.toaddress = memcpy.toaddress - B;

        if !fault then
            memcpy.cpysize = memcpy.cpysize - B;
            memcpy.stagecpysize = memcpy.stagecpysize - B;

UpdateCpyRegisters(memcpy, fault, copied);

if fault then
    if IsFault(memaddrdesc) then
        AArch64.Abort(memaddrdesc.vaddress, memaddrdesc.fault);
    else
        AccessDescriptor accdesc = if iswrite then waccdesc else raccdesc;
        HandleExternalAbort(memstatus, iswrite, memaddrdesc, B, accdesc);

if memcpy.stage == MOPSStage_Prologue then
    PSTATE.<N,Z,C,V> = memcpy.nzcv;
```

C6.2.95 CPYPRTWN, CPYMRTWN, CPYERTWN

Memory Copy, reads unprivileged, writes non-temporal. These instructions perform a memory copy. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYPRTWN, then CPYMRTWN, and then CPYERTWN.

CPYPRTWN performs some preconditioning of the arguments suitable for using the CPYMRTWN instruction, and performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYMRTWN performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYERTWN performs the last part of the memory copy.

Note

The inclusion of IMPLEMENTATION DEFINED amounts of memory copy allows some optimization of the size that can be performed.

For CPYPRTWN, the following saturation logic is applied:

If Xn<63:55> != 000000000, the copy size Xn is saturated to `0x007FFFFFFFFFFFFF`.

After that saturation logic is applied, the direction of the memory copy is based on the following algorithm:

If (Xs > Xd) && (Xd + saturated Xn) > Xs, then direction = forward

Elsif (Xs < Xd) && (Xs + saturated Xn) > Xd, then direction = backward

Else direction = IMPLEMENTATION DEFINED choice between forward and backward.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

After execution of CPYPRTWN, option A (which results in encoding PSTATE.C = 0):

- PSTATE.{N,Z,V} are set to {0,0,0}.
- If the copy is in the forward direction, then:
  - Xs holds the original Xs + saturated Xn.
  - Xd holds the original Xd + saturated Xn.
  - Xn holds -1* saturated Xn + an IMPLEMENTATION DEFINED number of bytes copied.
- If the copy is in the backward direction, then:
  - Xs and Xd are unchanged.
  - Xn holds the saturated value of Xn - an IMPLEMENTATION DEFINED number of bytes copied.

After execution of CPYPRTWN, option B (which results in encoding PSTATE.C = 1):

- If the copy is in the forward direction, then:
  - Xs holds the original Xs + an IMPLEMENTATION DEFINED number of bytes copied.
  - Xd holds the original Xd + an IMPLEMENTATION DEFINED number of bytes copied.
  - Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - PSTATE.{N,Z,V} are set to {0,0,0}.
- If the copy is in the backward direction, then:
  - Xs holds the original Xs + saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - Xd holds the original Xd + saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - PSTATE.{N,Z,V} are set to {1,0,0}.

For CPYMRTWN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number.
- If the copy is in the forward direction (Xn is a negative number), then:
  - Xn holds -1* the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the lowest address that the copy is copied from -Xn.
  - Xd holds the lowest address that the copy is made to -Xn.
  - At the end of the instruction, the value of Xn is written back with -1* the number of bytes remaining to be copied in the memory copy in total.
- If the copy is in the backward direction (Xn is a positive number), then:
  - Xn holds the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the highest address that the copy is copied from -Xn+1.
  - Xd holds the highest address that the copy is copied to -Xn+1.
  - At the end of the instruction, the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.

For CPYMRTWN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes to be copied in the memory copy in total.
- If the copy is in the forward direction (PSTATE.N == 0), then:
  - Xs holds the lowest address that the copy is copied from.
  - Xd holds the lowest address that the copy is copied to.
  - At the end of the instruction:
    - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
    - the value of Xs is written back with the lowest address that has not been copied from.
    - the value of Xd is written back with the lowest address that has not been copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
  - Xs holds the highest address that the copy is copied from +1.
  - Xd holds the highest address that the copy is copied to +1.
  - At the end of the instruction:
    - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
    - the value of Xs is written back with the highest address that has not been copied from +1.
    - the value of Xd is written back with the highest address that has not been copied to +1.

For CPYERTWN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number.
- If the copy is in the forward direction (Xn is a negative number), then:
  - Xn holds -1* the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the lowest address that the copy is copied from -Xn.
  - Xd holds the lowest address that the copy is made to -Xn.
  - At the end of the instruction, the value of Xn is written back with 0.
- If the copy is in the backward direction (Xn is a positive number), then:
  - Xn holds the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the highest address that the copy is copied from -Xn+1.

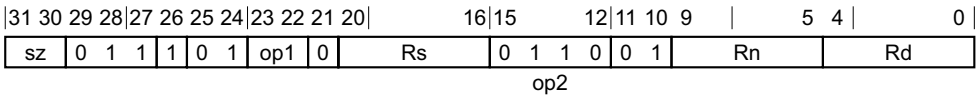
  - Xd holds the highest address that the copy is copied to -Xn+1.
  - At the end of the instruction, the value of Xn is written back with 0.

For CPYERTWN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes to be copied in the memory copy in total.
- If the copy is in the forward direction (PSTATE.N == 0), then:
  - Xs holds the lowest address that the copy is copied from.
  - Xd holds the lowest address that the copy is copied to.
  - At the end of the instruction:
    - the value of Xn is written back with 0.
    - the value of Xs is written back with the lowest address that has not been copied from.
    - the value of Xd is written back with the lowest address that has not been copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
  - Xs holds the highest address that the copy is copied from +1.
  - Xd holds the highest address that the copy is copied to +1.
  - At the end of the instruction:
    - the value of Xn is written back with 0.
    - the value of Xs is written back with the highest address that has not been copied from +1.
    - the value of Xd is written back with the highest address that has not been copied to +1.

Integer

(FEAT_MOPS)



Epilogue variant

Applies when `op1 == 10`.

```
CPYERTWN [<Xd>]!, [<Xs>]!, <Xn>!
```

Main variant

Applies when `op1 == 01`.

```
CPYMRTWN [<Xd>]!, [<Xs>]!, <Xn>!
```

Prologue variant

Applies when `op1 == 00`.

```
CPYPRTWN [<Xd>]!, [<Xs>]!, <Xn>!
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_MOPS) || sz != '00' then UNDEFINED;

CPYParams memcpy;
memcpy.d = UInt(Rd);
memcpy.s = UInt(Rs);
memcpy.n = UInt(Rn);
bits(4) options = op2;
```

```
boolean rnontemporal = options<3> == '1';
boolean wnontemporal = options<2> == '1';

case op1 of
    when '00' memcpy.stage = MOPSStage_Prologue;
    when '01' memcpy.stage = MOPSStage_Main;
    when '10' memcpy.stage = MOPSStage_Epilogue;
    otherwise SEE "Memory Copy and Memory Set";

CheckMOPSEnabled();

if (memcpy.s == memcpy.n || memcpy.s == memcpy.d || memcpy.n == memcpy.d || memcpy.d == 31 || memcpy.s
== 31 || memcpy.n == 31) then
    Constraint c = ConstrainUnpredictable(Unpredictable_MOPSOVERLAP31);
    assert c IN {Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNDEF UNDEFINED;
        when Constraint_NOP   EndOfInstruction();
```

Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*, and particularly *Memory Copy and Memory Set CPY**.

Assembler symbols

<Xd> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the "Rd" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.

<Xs> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the "Rs" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the "Rs" field.

<Xn> For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero at the end of the instruction, encoded in the "Rn" field.

For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the "Rn" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the "Rn" field.

Operation

```
constant integer N = MaxBlockSizeCopiedBytes();
bits(8*N) readdata;

memcpy.nzcv = PSTATE.<N,Z,C,V>;
memcpy.toaddress = X[memcpy.d, 64];
memcpy.fromaddress = X[memcpy.s, 64];

if memcpy.stage == MOPSStage_Prologue then
    memcpy.cpysize = UInt(X[memcpy.n, 64]);
else
    memcpy.cpysize = SInt(X[memcpy.n, 64]);

memcpy.implements_option_a = CPYOptionA();
```

```
boolean rprivileged = if options<1> == '1' then AArch64.IsUnprivAccessPriv() else PSTATE.EL != EL0;
boolean wprivileged = if options<0> == '1' then AArch64.IsUnprivAccessPriv() else PSTATE.EL != EL0;

AccessDescriptor raccdesc = CreateAccDescMOPS(MemOp_LOAD, rprivileged, rnontemporal);
AccessDescriptor waccdesc = CreateAccDescMOPS(MemOp_STORE, wprivileged, wnontemporal);

if memcpy.stage == MOPSStage_Prologue then
    if memcpy.cpysize > 0x007FFFFFFFFFFFF then
        memcpy.cpysize = 0x007FFFFFFFFFFFF;

    memcpy.forward = IsMemCpyForward(memcpy);

    if memcpy.implements_option_a then
        memcpy.nzcv = '0000';
        if memcpy.forward then
            // Copy in the forward direction offsets the arguments.
            memcpy.toaddress = memcpy.toaddress + memcpy.cpysize;
            memcpy.fromaddress = memcpy.fromaddress + memcpy.cpysize;
            memcpy.cpysize = 0 - memcpy.cpysize;
    else
        if !memcpy.forward then
            // Copy in the reverse direction offsets the arguments.
            memcpy.toaddress = memcpy.toaddress + memcpy.cpysize;
            memcpy.fromaddress = memcpy.fromaddress + memcpy.cpysize;
            memcpy.nzcv = '1010';
        else
            memcpy.nzcv = '0010';

memcpy.stagecpysize = MemCpyStageSize(memcpy);

if memcpy.stage != MOPSStage_Prologue then
    memcpy.forward = memcpy.cpysize < 0 || (!memcpy.implements_option_a && memcpy.nzcv<3> == '0');
    CheckMemCpyParams(memcpy, options);

integer copied;
boolean iswrite;
AddressDescriptor memaddrdesc;
PhysMemRetStatus memstatus;
boolean fault = FALSE;
integer B;

if memcpy.implements_option_a then
    while memcpy.stagecpysize != 0 && !fault do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(memcpy);

        if memcpy.forward then
            assert B <= -1 * memcpy.stagecpysize;
            (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress + memcpy.cpysize,
memcpy.fromaddress + memcpy.cpysize, memcpy.forward, B, raccdesc, waccdesc);
            if copied != B then
                fault = TRUE;
            else
                memcpy.cpysize = memcpy.cpysize + B;
                memcpy.stagecpysize = memcpy.stagecpysize + B;

        else
            assert B <= memcpy.stagecpysize;
            memcpy.cpysize = memcpy.cpysize - B;
            memcpy.stagecpysize = memcpy.stagecpysize - B;

            (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress + memcpy.cpysize,
memcpy.fromaddress + memcpy.cpysize, memcpy.forward, B, raccdesc, waccdesc);
            if copied != B then
                fault = TRUE;
                memcpy.cpysize = memcpy.cpysize + B;
                memcpy.stagecpysize = memcpy.stagecpysize + B;
```

```
else
    while memcpy.stagecpysize > 0 && !fault do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(memcpy);
        assert B <= memcpy.stagecpysize;

        if memcpy.forward then
            (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress,
memcpy.fromaddress, memcpy.forward, B, raccdesc, waccdesc);
            if copied != B then
                fault = TRUE;
            else
                memcpy.fromaddress = memcpy.fromaddress + B;
                memcpy.toaddress = memcpy.toaddress + B;
        else
            (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress - B,
memcpy.fromaddress - B, memcpy.forward, B, raccdesc, waccdesc);

            if copied != B then
                fault = TRUE;
            else
                memcpy.fromaddress = memcpy.fromaddress - B;
                memcpy.toaddress = memcpy.toaddress - B;

        if !fault then
            memcpy.cpysize = memcpy.cpysize - B;
            memcpy.stagecpysize = memcpy.stagecpysize - B;

UpdateCpyRegisters(memcpy, fault, copied);

if fault then
    if IsFault(memaddrdesc) then
        AArch64.Abort(memaddrdesc.vaddress, memaddrdesc.fault);
    else
        AccessDescriptor accdesc = if iswrite then waccdesc else raccdesc;
        HandleExternalAbort(memstatus, iswrite, memaddrdesc, B, accdesc);

if memcpy.stage == MOPSStage_Prologue then
    PSTATE.<N,Z,C,V> = memcpy.nzcv;
```

C6.2.96 CPYPT, CPYMT, CPYET

Memory Copy, reads and writes unprivileged. These instructions perform a memory copy. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYPT, then CPYMT, and then CPYET.

CPYPT performs some preconditioning of the arguments suitable for using the CPYMT instruction, and performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYMT performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYET performs the last part of the memory copy.

Note

The inclusion of IMPLEMENTATION DEFINED amounts of memory copy allows some optimization of the size that can be performed.

For CPYPT, the following saturation logic is applied:

If Xn<63:55> != 000000000, the copy size Xn is saturated to `0x007FFFFFFFFFFFFF`.

After that saturation logic is applied, the direction of the memory copy is based on the following algorithm:

If (Xs > Xd) && (Xd + saturated Xn) > Xs, then direction = forward

Elsif (Xs < Xd) && (Xs + saturated Xn) > Xd, then direction = backward

Else direction = IMPLEMENTATION DEFINED choice between forward and backward.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

After execution of CPYPT, option A (which results in encoding PSTATE.C = 0):

- PSTATE.{N,Z,V} are set to {0,0,0}.
- If the copy is in the forward direction, then:
  - Xs holds the original Xs + saturated Xn.
  - Xd holds the original Xd + saturated Xn.
  - Xn holds -1* saturated Xn + an IMPLEMENTATION DEFINED number of bytes copied.
- If the copy is in the backward direction, then:
  - Xs and Xd are unchanged.
  - Xn holds the saturated value of Xn - an IMPLEMENTATION DEFINED number of bytes copied.

After execution of CPYPT, option B (which results in encoding PSTATE.C = 1):

- If the copy is in the forward direction, then:
  - Xs holds the original Xs + an IMPLEMENTATION DEFINED number of bytes copied.
  - Xd holds the original Xd + an IMPLEMENTATION DEFINED number of bytes copied.
  - Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - PSTATE.{N,Z,V} are set to {0,0,0}.
- If the copy is in the backward direction, then:
  - Xs holds the original Xs + saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - Xd holds the original Xd + saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - PSTATE.{N,Z,V} are set to {1,0,0}.

For CPYMT, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number.
- If the copy is in the forward direction (Xn is a negative number), then:
  - Xn holds -1* the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the lowest address that the copy is copied from -Xn.
  - Xd holds the lowest address that the copy is made to -Xn.
  - At the end of the instruction, the value of Xn is written back with -1* the number of bytes remaining to be copied in the memory copy in total.
- If the copy is in the backward direction (Xn is a positive number), then:
  - Xn holds the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the highest address that the copy is copied from -Xn+1.
  - Xd holds the highest address that the copy is copied to -Xn+1.
  - At the end of the instruction, the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.

For CPYMT, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes to be copied in the memory copy in total.
- If the copy is in the forward direction (PSTATE.N == 0), then:
  - Xs holds the lowest address that the copy is copied from.
  - Xd holds the lowest address that the copy is copied to.
  - At the end of the instruction:
    - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
    - the value of Xs is written back with the lowest address that has not been copied from.
    - the value of Xd is written back with the lowest address that has not been copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
  - Xs holds the highest address that the copy is copied from +1.
  - Xd holds the highest address that the copy is copied to +1.
  - At the end of the instruction:
    - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
    - the value of Xs is written back with the highest address that has not been copied from +1.
    - the value of Xd is written back with the highest address that has not been copied to +1.

For CPYET, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number.
- If the copy is in the forward direction (Xn is a negative number), then:
  - Xn holds -1* the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the lowest address that the copy is copied from -Xn.
  - Xd holds the lowest address that the copy is made to -Xn.
  - At the end of the instruction, the value of Xn is written back with 0.
- If the copy is in the backward direction (Xn is a positive number), then:
  - Xn holds the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the highest address that the copy is copied from -Xn+1.

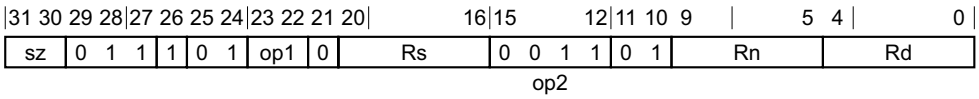
- Xd holds the highest address that the copy is copied to -Xn+1.
- At the end of the instruction, the value of Xn is written back with 0.

For CPYET, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes to be copied in the memory copy in total.
- If the copy is in the forward direction (PSTATE.N == 0), then:
  - Xs holds the lowest address that the copy is copied from.
  - Xd holds the lowest address that the copy is copied to.
  - At the end of the instruction:
    - the value of Xn is written back with 0.
    - the value of Xs is written back with the lowest address that has not been copied from.
    - the value of Xd is written back with the lowest address that has not been copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
  - Xs holds the highest address that the copy is copied from +1.
  - Xd holds the highest address that the copy is copied to +1.
  - At the end of the instruction:
    - the value of Xn is written back with 0.
    - the value of Xs is written back with the highest address that has not been copied from +1.
    - the value of Xd is written back with the highest address that has not been copied to +1.

Integer

(FEAT_MOPS)



Epilogue variant

Applies when `op1 == 10`.

```
CPYET [<Xd>]!, [<Xs>]!, <Xn>!
```

Main variant

Applies when `op1 == 01`.

```
CPYMT [<Xd>]!, [<Xs>]!, <Xn>!
```

Prologue variant

Applies when `op1 == 00`.

```
CPYPT [<Xd>]!, [<Xs>]!, <Xn>!
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_MOPS) || sz != '00' then UNDEFINED;

CPYParams memcpy;
memcpy.d = UInt(Rd);
memcpy.s = UInt(Rs);
memcpy.n = UInt(Rn);
bits(4) options = op2;
```

```
boolean rnontemporal = options<3> == '1';
boolean wnontemporal = options<2> == '1';

case op1 of
    when '00' memcpy.stage = MOPSStage_Prologue;
    when '01' memcpy.stage = MOPSStage_Main;
    when '10' memcpy.stage = MOPSStage_Epilogue;
    otherwise SEE "Memory Copy and Memory Set";

CheckMOPSEnabled();

if (memcpy.s == memcpy.n || memcpy.s == memcpy.d || memcpy.n == memcpy.d || memcpy.d == 31 || memcpy.s
== 31 || memcpy.n == 31) then
    Constraint c = ConstrainUnpredictable(Unpredictable_MOPSOVERLAP31);
    assert c IN {Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNDEF UNDEFINED;
        when Constraint_NOP   EndOfInstruction();
```

Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*, and particularly *Memory Copy and Memory Set CPY**.

Assembler symbols

<Xd> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the "Rd" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.

<Xs> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the "Rs" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the "Rs" field.

<Xn> For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero at the end of the instruction, encoded in the "Rn" field.

For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the "Rn" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the "Rn" field.

Operation

```
constant integer N = MaxBlockSizeCopiedBytes();
bits(8*N) readdata;

memcpy.nzcv = PSTATE.<N,Z,C,V>;
memcpy.toaddress = X[memcpy.d, 64];
memcpy.fromaddress = X[memcpy.s, 64];

if memcpy.stage == MOPSStage_Prologue then
    memcpy.cpysize = UInt(X[memcpy.n, 64]);
else
    memcpy.cpysize = SInt(X[memcpy.n, 64]);

memcpy.implements_option_a = CPYOptionA();
```

```
boolean rprivileged = if options<1> == '1' then AArch64.IsUnprivAccessPriv() else PSTATE.EL != EL0;
boolean wprivileged = if options<0> == '1' then AArch64.IsUnprivAccessPriv() else PSTATE.EL != EL0;

AccessDescriptor raccdesc = CreateAccDescMOPS(MemOp_LOAD, rprivileged, rnontemporal);
AccessDescriptor waccdesc = CreateAccDescMOPS(MemOp_STORE, wprivileged, wnontemporal);

if memcpy.stage == MOPSStage_Prologue then
    if memcpy.cpysize > 0x007FFFFFFFFFFFF then
        memcpy.cpysize = 0x007FFFFFFFFFFFF;

    memcpy.forward = IsMemCpyForward(memcpy);

    if memcpy.implements_option_a then
        memcpy.nzcv = '0000';
        if memcpy.forward then
            // Copy in the forward direction offsets the arguments.
            memcpy.toaddress = memcpy.toaddress + memcpy.cpysize;
            memcpy.fromaddress = memcpy.fromaddress + memcpy.cpysize;
            memcpy.cpysize = 0 - memcpy.cpysize;
    else
        if !memcpy.forward then
            // Copy in the reverse direction offsets the arguments.
            memcpy.toaddress = memcpy.toaddress + memcpy.cpysize;
            memcpy.fromaddress = memcpy.fromaddress + memcpy.cpysize;
            memcpy.nzcv = '1010';
        else
            memcpy.nzcv = '0010';

memcpy.stagecpysize = MemCpyStageSize(memcpy);

if memcpy.stage != MOPSStage_Prologue then
    memcpy.forward = memcpy.cpysize < 0 || (!memcpy.implements_option_a && memcpy.nzcv<3> == '0');
    CheckMemCpyParams(memcpy, options);

integer copied;
boolean iswrite;
AddressDescriptor memaddrdesc;
PhysMemRetStatus memstatus;
boolean fault = FALSE;
integer B;

if memcpy.implements_option_a then
    while memcpy.stagecpysize != 0 && !fault do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(memcpy);

        if memcpy.forward then
            assert B <= -1 * memcpy.stagecpysize;
            (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress + memcpy.cpysize,
memcpy.fromaddress + memcpy.cpysize, memcpy.forward, B, raccdesc, waccdesc);
            if copied != B then
                fault = TRUE;
            else
                memcpy.cpysize = memcpy.cpysize + B;
                memcpy.stagecpysize = memcpy.stagecpysize + B;

        else
            assert B <= memcpy.stagecpysize;
            memcpy.cpysize = memcpy.cpysize - B;
            memcpy.stagecpysize = memcpy.stagecpysize - B;

            (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress + memcpy.cpysize,
memcpy.fromaddress + memcpy.cpysize, memcpy.forward, B, raccdesc, waccdesc);
            if copied != B then
                fault = TRUE;
                memcpy.cpysize = memcpy.cpysize + B;
                memcpy.stagecpysize = memcpy.stagecpysize + B;
```

```
else
    while memcpy.stagecpysize > 0 && !fault do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(memcpy);
        assert B <= memcpy.stagecpysize;

        if memcpy.forward then
            (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress,
memcpy.fromaddress, memcpy.forward, B, raccdesc, waccdesc);
            if copied != B then
                fault = TRUE;
            else
                memcpy.fromaddress = memcpy.fromaddress + B;
                memcpy.toaddress = memcpy.toaddress + B;
        else
            (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress - B,
memcpy.fromaddress - B, memcpy.forward, B, raccdesc, waccdesc);

            if copied != B then
                fault = TRUE;
            else
                memcpy.fromaddress = memcpy.fromaddress - B;
                memcpy.toaddress = memcpy.toaddress - B;

        if !fault then
            memcpy.cpysize = memcpy.cpysize - B;
            memcpy.stagecpysize = memcpy.stagecpysize - B;

UpdateCpyRegisters(memcpy, fault, copied);

if fault then
    if IsFault(memaddrdesc) then
        AArch64.Abort(memaddrdesc.vaddress, memaddrdesc.fault);
    else
        AccessDescriptor accdesc = if iswrite then waccdesc else raccdesc;
        HandleExternalAbort(memstatus, iswrite, memaddrdesc, B, accdesc);

if memcpy.stage == MOPSStage_Prologue then
    PSTATE.<N,Z,C,V> = memcpy.nzcv;
```

C6.2.97 CPYPTN, CPYMTN, CPYETN

Memory Copy, reads and writes unprivileged and non-temporal. These instructions perform a memory copy. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYPTN, then CPYMTN, and then CPYETN.

CPYPTN performs some preconditioning of the arguments suitable for using the CPYMTN instruction, and performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYMTN performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYETN performs the last part of the memory copy.

Note

The inclusion of IMPLEMENTATION DEFINED amounts of memory copy allows some optimization of the size that can be performed.

For CPYPTN, the following saturation logic is applied:

If Xn<63:55> != 000000000, the copy size Xn is saturated to `0x007FFFFFFFFFFFF`.

After that saturation logic is applied, the direction of the memory copy is based on the following algorithm:

If (Xs > Xd) && (Xd + saturated Xn) > Xs, then direction = forward

Elsif (Xs < Xd) && (Xs + saturated Xn) > Xd, then direction = backward

Else direction = IMPLEMENTATION DEFINED choice between forward and backward.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

After execution of CPYPTN, option A (which results in encoding PSTATE.C = 0):

- PSTATE.{N,Z,V} are set to {0,0,0}.
- If the copy is in the forward direction, then:
  - Xs holds the original Xs + saturated Xn.
  - Xd holds the original Xd + saturated Xn.
  - Xn holds -1* saturated Xn + an IMPLEMENTATION DEFINED number of bytes copied.
- If the copy is in the backward direction, then:
  - Xs and Xd are unchanged.
  - Xn holds the saturated value of Xn - an IMPLEMENTATION DEFINED number of bytes copied.

After execution of CPYPTN, option B (which results in encoding PSTATE.C = 1):

- If the copy is in the forward direction, then:
  - Xs holds the original Xs + an IMPLEMENTATION DEFINED number of bytes copied.
  - Xd holds the original Xd + an IMPLEMENTATION DEFINED number of bytes copied.
  - Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - PSTATE.{N,Z,V} are set to {0,0,0}.
- If the copy is in the backward direction, then:
  - Xs holds the original Xs + saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - Xd holds the original Xd + saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - PSTATE.{N,Z,V} are set to {1,0,0}.

For CPYMTN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number.
- If the copy is in the forward direction (Xn is a negative number), then:
  - Xn holds -1* the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the lowest address that the copy is copied from -Xn.
  - Xd holds the lowest address that the copy is made to -Xn.
  - At the end of the instruction, the value of Xn is written back with -1* the number of bytes remaining to be copied in the memory copy in total.
- If the copy is in the backward direction (Xn is a positive number), then:
  - Xn holds the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the highest address that the copy is copied from -Xn+1.
  - Xd holds the highest address that the copy is copied to -Xn+1.
  - At the end of the instruction, the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.

For CPYMTN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes to be copied in the memory copy in total.
- If the copy is in the forward direction (PSTATE.N == 0), then:
  - Xs holds the lowest address that the copy is copied from.
  - Xd holds the lowest address that the copy is copied to.
  - At the end of the instruction:
    - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
    - the value of Xs is written back with the lowest address that has not been copied from.
    - the value of Xd is written back with the lowest address that has not been copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
  - Xs holds the highest address that the copy is copied from +1.
  - Xd holds the highest address that the copy is copied to +1.
  - At the end of the instruction:
    - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
    - the value of Xs is written back with the highest address that has not been copied from +1.
    - the value of Xd is written back with the highest address that has not been copied to +1.

For CPYETN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number.
- If the copy is in the forward direction (Xn is a negative number), then:
  - Xn holds -1* the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the lowest address that the copy is copied from -Xn.
  - Xd holds the lowest address that the copy is made to -Xn.
  - At the end of the instruction, the value of Xn is written back with 0.
- If the copy is in the backward direction (Xn is a positive number), then:
  - Xn holds the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the highest address that the copy is copied from -Xn+1.

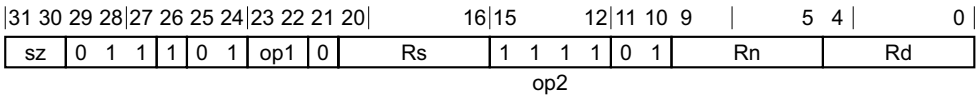
— Xd holds the highest address that the copy is copied to -Xn+1.
— At the end of the instruction, the value of Xn is written back with 0.

For CPYETN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes to be copied in the memory copy in total.
- If the copy is in the forward direction (PSTATE.N == 0), then:
  — Xs holds the lowest address that the copy is copied from.
  — Xd holds the lowest address that the copy is copied to.
  — At the end of the instruction:
    — the value of Xn is written back with 0.
    — the value of Xs is written back with the lowest address that has not been copied from.
    — the value of Xd is written back with the lowest address that has not been copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
  — Xs holds the highest address that the copy is copied from +1.
  — Xd holds the highest address that the copy is copied to +1.
  — At the end of the instruction:
    — the value of Xn is written back with 0.
    — the value of Xs is written back with the highest address that has not been copied from +1.
    — the value of Xd is written back with the highest address that has not been copied to +1.

Integer

(FEAT_MOPS)



Epilogue variant

Applies when `op1 == 10`.

```
CPYETN [<Xd>]!, [<Xs>]!, <Xn>!
```

Main variant

Applies when `op1 == 01`.

```
CPYMTN [<Xd>]!, [<Xs>]!, <Xn>!
```

Prologue variant

Applies when `op1 == 00`.

```
CPYPTN [<Xd>]!, [<Xs>]!, <Xn>!
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_MOPS) || sz != '00' then UNDEFINED;

CPYParams memcpy;
memcpy.d = UInt(Rd);
memcpy.s = UInt(Rs);
memcpy.n = UInt(Rn);
bits(4) options = op2;
```

```
boolean rnontemporal = options<3> == '1';
boolean wnontemporal = options<2> == '1';

case op1 of
    when '00' memcpy.stage = MOPSStage_Prologue;
    when '01' memcpy.stage = MOPSStage_Main;
    when '10' memcpy.stage = MOPSStage_Epilogue;
    otherwise SEE "Memory Copy and Memory Set";

CheckMOPSEnabled();

if (memcpy.s == memcpy.n || memcpy.s == memcpy.d || memcpy.n == memcpy.d || memcpy.d == 31 || memcpy.s
== 31 || memcpy.n == 31) then
    Constraint c = ConstrainUnpredictable(Unpredictable_MOPSOVERLAP31);
    assert c IN {Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNDEF UNDEFINED;
        when Constraint_NOP   EndOfInstruction();
```

Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*, and particularly *Memory Copy and Memory Set CPY**.

Assembler symbols

<Xd> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the "Rd" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.

<Xs> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the "Rs" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the "Rs" field.

<Xn> For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero at the end of the instruction, encoded in the "Rn" field.

For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the "Rn" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the "Rn" field.

Operation

```
constant integer N = MaxBlockSizeCopiedBytes();
bits(8*N) readdata;

memcpy.nzcv = PSTATE.<N,Z,C,V>;
memcpy.toaddress = X[memcpy.d, 64];
memcpy.fromaddress = X[memcpy.s, 64];

if memcpy.stage == MOPSStage_Prologue then
    memcpy.cpysize = UInt(X[memcpy.n, 64]);
else
    memcpy.cpysize = SInt(X[memcpy.n, 64]);

memcpy.implements_option_a = CPYOptionA();
```

```
boolean rprivileged = if options<1> == '1' then AArch64.IsUnprivAccessPriv() else PSTATE.EL != EL0;
boolean wprivileged = if options<0> == '1' then AArch64.IsUnprivAccessPriv() else PSTATE.EL != EL0;

AccessDescriptor raccdesc = CreateAccDescMOPS(MemOp_LOAD, rprivileged, rnontemporal);
AccessDescriptor waccdesc = CreateAccDescMOPS(MemOp_STORE, wprivileged, wnontemporal);

if memcpy.stage == MOPSStage_Prologue then
    if memcpy.cpysize > 0x007FFFFFFFFFFFF then
        memcpy.cpysize = 0x007FFFFFFFFFFFF;

    memcpy.forward = IsMemCpyForward(memcpy);

    if memcpy.implements_option_a then
        memcpy.nzcv = '0000';
        if memcpy.forward then
            // Copy in the forward direction offsets the arguments.
            memcpy.toaddress = memcpy.toaddress + memcpy.cpysize;
            memcpy.fromaddress = memcpy.fromaddress + memcpy.cpysize;
            memcpy.cpysize = 0 - memcpy.cpysize;
    else
        if !memcpy.forward then
            // Copy in the reverse direction offsets the arguments.
            memcpy.toaddress = memcpy.toaddress + memcpy.cpysize;
            memcpy.fromaddress = memcpy.fromaddress + memcpy.cpysize;
            memcpy.nzcv = '1010';
        else
            memcpy.nzcv = '0010';

memcpy.stagecpysize = MemCpyStageSize(memcpy);

if memcpy.stage != MOPSStage_Prologue then
    memcpy.forward = memcpy.cpysize < 0 || (!memcpy.implements_option_a && memcpy.nzcv<3> == '0');
    CheckMemCpyParams(memcpy, options);

integer copied;
boolean iswrite;
AddressDescriptor memaddrdesc;
PhysMemRetStatus memstatus;
boolean fault = FALSE;
integer B;

if memcpy.implements_option_a then
    while memcpy.stagecpysize != 0 && !fault do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(memcpy);

        if memcpy.forward then
            assert B <= -1 * memcpy.stagecpysize;
            (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress + memcpy.cpysize,
memcpy.fromaddress + memcpy.cpysize, memcpy.forward, B, raccdesc, waccdesc);
            if copied != B then
                fault = TRUE;
            else
                memcpy.cpysize = memcpy.cpysize + B;
                memcpy.stagecpysize = memcpy.stagecpysize + B;

        else
            assert B <= memcpy.stagecpysize;
            memcpy.cpysize = memcpy.cpysize - B;
            memcpy.stagecpysize = memcpy.stagecpysize - B;

            (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress + memcpy.cpysize,
memcpy.fromaddress + memcpy.cpysize, memcpy.forward, B, raccdesc, waccdesc);
            if copied != B then
                fault = TRUE;
                memcpy.cpysize = memcpy.cpysize + B;
                memcpy.stagecpysize = memcpy.stagecpysize + B;
```

```
else
    while memcpy.stagecpysize > 0 && !fault do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(memcpy);
        assert B <= memcpy.stagecpysize;

        if memcpy.forward then
            (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress,
memcpy.fromaddress, memcpy.forward, B, raccdesc, waccdesc);
            if copied != B then
                fault = TRUE;
            else
                memcpy.fromaddress = memcpy.fromaddress + B;
                memcpy.toaddress = memcpy.toaddress + B;
        else
            (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress - B,
memcpy.fromaddress - B, memcpy.forward, B, raccdesc, waccdesc);

            if copied != B then
                fault = TRUE;
            else
                memcpy.fromaddress = memcpy.fromaddress - B;
                memcpy.toaddress = memcpy.toaddress - B;

        if !fault then
            memcpy.cpysize = memcpy.cpysize - B;
            memcpy.stagecpysize = memcpy.stagecpysize - B;

UpdateCpyRegisters(memcpy, fault, copied);

if fault then
    if IsFault(memaddrdesc) then
        AArch64.Abort(memaddrdesc.vaddress, memaddrdesc.fault);
    else
        AccessDescriptor accdesc = if iswrite then waccdesc else raccdesc;
        HandleExternalAbort(memstatus, iswrite, memaddrdesc, B, accdesc);

if memcpy.stage == MOPSStage_Prologue then
    PSTATE.<N,Z,C,V> = memcpy.nzcv;
```

C6.2.98 CPYPTRN, CPYMTRN, CPYETRN

Memory Copy, reads and writes unprivileged, reads non-temporal. These instructions perform a memory copy. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYPTRN, then CPYMTRN, and then CPYETRN.

CPYPTRN performs some preconditioning of the arguments suitable for using the CPYMTRN instruction, and performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYMTRN performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYETRN performs the last part of the memory copy.

Note

The inclusion of IMPLEMENTATION DEFINED amounts of memory copy allows some optimization of the size that can be performed.

For CPYPTRN, the following saturation logic is applied:

If Xn<63:55> != 000000000, the copy size Xn is saturated to `0x007FFFFFFFFFFFF`.

After that saturation logic is applied, the direction of the memory copy is based on the following algorithm:

If (Xs > Xd) && (Xd + saturated Xn) > Xs, then direction = forward

Elsif (Xs < Xd) && (Xs + saturated Xn) > Xd, then direction = backward

Else direction = IMPLEMENTATION DEFINED choice between forward and backward.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

After execution of CPYPTRN, option A (which results in encoding PSTATE.C = 0):

- PSTATE.{N,Z,V} are set to {0,0,0}.
- If the copy is in the forward direction, then:
  - Xs holds the original Xs + saturated Xn.
  - Xd holds the original Xd + saturated Xn.
  - Xn holds -1* saturated Xn + an IMPLEMENTATION DEFINED number of bytes copied.
- If the copy is in the backward direction, then:
  - Xs and Xd are unchanged.
  - Xn holds the saturated value of Xn - an IMPLEMENTATION DEFINED number of bytes copied.

After execution of CPYPTRN, option B (which results in encoding PSTATE.C = 1):

- If the copy is in the forward direction, then:
  - Xs holds the original Xs + an IMPLEMENTATION DEFINED number of bytes copied.
  - Xd holds the original Xd + an IMPLEMENTATION DEFINED number of bytes copied.
  - Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - PSTATE.{N,Z,V} are set to {0,0,0}.
- If the copy is in the backward direction, then:
  - Xs holds the original Xs + saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - Xd holds the original Xd + saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - PSTATE.{N,Z,V} are set to {1,0,0}.

For CPYMTRN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number.
- If the copy is in the forward direction (Xn is a negative number), then:
  - Xn holds -1* the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the lowest address that the copy is copied from -Xn.
  - Xd holds the lowest address that the copy is made to -Xn.
  - At the end of the instruction, the value of Xn is written back with -1* the number of bytes remaining to be copied in the memory copy in total.
- If the copy is in the backward direction (Xn is a positive number), then:
  - Xn holds the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the highest address that the copy is copied from -Xn+1.
  - Xd holds the highest address that the copy is copied to -Xn+1.
  - At the end of the instruction, the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.

For CPYMTRN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes to be copied in the memory copy in total.
- If the copy is in the forward direction (PSTATE.N == 0), then:
  - Xs holds the lowest address that the copy is copied from.
  - Xd holds the lowest address that the copy is copied to.
  - At the end of the instruction:
    - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
    - the value of Xs is written back with the lowest address that has not been copied from.
    - the value of Xd is written back with the lowest address that has not been copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
  - Xs holds the highest address that the copy is copied from +1.
  - Xd holds the highest address that the copy is copied to +1.
  - At the end of the instruction:
    - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
    - the value of Xs is written back with the highest address that has not been copied from +1.
    - the value of Xd is written back with the highest address that has not been copied to +1.

For CPYETRN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number.
- If the copy is in the forward direction (Xn is a negative number), then:
  - Xn holds -1* the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the lowest address that the copy is copied from -Xn.
  - Xd holds the lowest address that the copy is made to -Xn.
  - At the end of the instruction, the value of Xn is written back with 0.
- If the copy is in the backward direction (Xn is a positive number), then:
  - Xn holds the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the highest address that the copy is copied from -Xn+1.

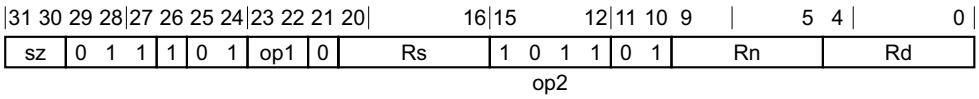
— Xd holds the highest address that the copy is copied to -Xn+1.
— At the end of the instruction, the value of Xn is written back with 0.

For CPYETRN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes to be copied in the memory copy in total.
- If the copy is in the forward direction (PSTATE.N == 0), then:
  — Xs holds the lowest address that the copy is copied from.
  — Xd holds the lowest address that the copy is copied to.
  — At the end of the instruction:
    — the value of Xn is written back with 0.
    — the value of Xs is written back with the lowest address that has not been copied from.
    — the value of Xd is written back with the lowest address that has not been copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
  — Xs holds the highest address that the copy is copied from +1.
  — Xd holds the highest address that the copy is copied to +1.
  — At the end of the instruction:
    — the value of Xn is written back with 0.
    — the value of Xs is written back with the highest address that has not been copied from +1.
    — the value of Xd is written back with the highest address that has not been copied to +1.

Integer

(FEAT_MOPS)



Epilogue variant

Applies when `op1 == 10`.

```
CPYETRN [<Xd>]!, [<Xs>]!, <Xn>!
```

Main variant

Applies when `op1 == 01`.

```
CPYMTRN [<Xd>]!, [<Xs>]!, <Xn>!
```

Prologue variant

Applies when `op1 == 00`.

```
CPYPTRN [<Xd>]!, [<Xs>]!, <Xn>!
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_MOPS) || sz != '00' then UNDEFINED;

CPYParams memcpy;
memcpy.d = UInt(Rd);
memcpy.s = UInt(Rs);
memcpy.n = UInt(Rn);
bits(4) options = op2;
```

```
boolean rnontemporal = options<3> == '1';
boolean wnontemporal = options<2> == '1';

case op1 of
    when '00' memcpy.stage = MOPSStage_Prologue;
    when '01' memcpy.stage = MOPSStage_Main;
    when '10' memcpy.stage = MOPSStage_Epilogue;
    otherwise SEE "Memory Copy and Memory Set";

CheckMOPSEnabled();

if (memcpy.s == memcpy.n || memcpy.s == memcpy.d || memcpy.n == memcpy.d || memcpy.d == 31 || memcpy.s
== 31 || memcpy.n == 31) then
    Constraint c = ConstrainUnpredictable(Unpredictable_MOPSOVERLAP31);
    assert c IN {Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNDEF UNDEFINED;
        when Constraint_NOP   EndOfInstruction();
```

Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*, and particularly *Memory Copy and Memory Set CPY**.

Assembler symbols

<Xd> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the "Rd" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.

<Xs> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the "Rs" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the "Rs" field.

<Xn> For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero at the end of the instruction, encoded in the "Rn" field.

For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the "Rn" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the "Rn" field.

Operation

```
constant integer N = MaxBlockSizeCopiedBytes();
bits(8*N) readdata;

memcpy.nzcv = PSTATE.<N,Z,C,V>;
memcpy.toaddress = X[memcpy.d, 64];
memcpy.fromaddress = X[memcpy.s, 64];

if memcpy.stage == MOPSStage_Prologue then
    memcpy.cpysize = UInt(X[memcpy.n, 64]);
else
    memcpy.cpysize = SInt(X[memcpy.n, 64]);

memcpy.implements_option_a = CPYOptionA();
```

```
boolean rprivileged = if options<1> == '1' then AArch64.IsUnprivAccessPriv() else PSTATE.EL != EL0;
boolean wprivileged = if options<0> == '1' then AArch64.IsUnprivAccessPriv() else PSTATE.EL != EL0;

AccessDescriptor raccdesc = CreateAccDescMOPS(MemOp_LOAD, rprivileged, rnontemporal);
AccessDescriptor waccdesc = CreateAccDescMOPS(MemOp_STORE, wprivileged, wnontemporal);

if memcpy.stage == MOPSStage_Prologue then
    if memcpy.cpysize > 0x007FFFFFFFFFFFF then
        memcpy.cpysize = 0x007FFFFFFFFFFFF;

    memcpy.forward = IsMemCpyForward(memcpy);

    if memcpy.implements_option_a then
        memcpy.nzcv = '0000';
        if memcpy.forward then
            // Copy in the forward direction offsets the arguments.
            memcpy.toaddress = memcpy.toaddress + memcpy.cpysize;
            memcpy.fromaddress = memcpy.fromaddress + memcpy.cpysize;
            memcpy.cpysize = 0 - memcpy.cpysize;
    else
        if !memcpy.forward then
            // Copy in the reverse direction offsets the arguments.
            memcpy.toaddress = memcpy.toaddress + memcpy.cpysize;
            memcpy.fromaddress = memcpy.fromaddress + memcpy.cpysize;
            memcpy.nzcv = '1010';
        else
            memcpy.nzcv = '0010';

memcpy.stagecpysize = MemCpyStageSize(memcpy);

if memcpy.stage != MOPSStage_Prologue then
    memcpy.forward = memcpy.cpysize < 0 || (!memcpy.implements_option_a && memcpy.nzcv<3> == '0');
    CheckMemCpyParams(memcpy, options);

integer copied;
boolean iswrite;
AddressDescriptor memaddrdesc;
PhysMemRetStatus memstatus;
boolean fault = FALSE;
integer B;

if memcpy.implements_option_a then
    while memcpy.stagecpysize != 0 && !fault do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(memcpy);

        if memcpy.forward then
            assert B <= -1 * memcpy.stagecpysize;
            (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress + memcpy.cpysize,
memcpy.fromaddress + memcpy.cpysize, memcpy.forward, B, raccdesc, waccdesc);
            if copied != B then
                fault = TRUE;
            else
                memcpy.cpysize = memcpy.cpysize + B;
                memcpy.stagecpysize = memcpy.stagecpysize + B;

        else
            assert B <= memcpy.stagecpysize;
            memcpy.cpysize = memcpy.cpysize - B;
            memcpy.stagecpysize = memcpy.stagecpysize - B;

            (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress + memcpy.cpysize,
memcpy.fromaddress + memcpy.cpysize, memcpy.forward, B, raccdesc, waccdesc);
            if copied != B then
                fault = TRUE;
                memcpy.cpysize = memcpy.cpysize + B;
                memcpy.stagecpysize = memcpy.stagecpysize + B;
```

```
else
    while memcpy.stagecpysize > 0 && !fault do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(memcpy);
        assert B <= memcpy.stagecpysize;

        if memcpy.forward then
            (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress,
memcpy.fromaddress, memcpy.forward, B, raccdesc, waccdesc);
            if copied != B then
                fault = TRUE;
            else
                memcpy.fromaddress = memcpy.fromaddress + B;
                memcpy.toaddress = memcpy.toaddress + B;
        else
            (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress - B,
memcpy.fromaddress - B, memcpy.forward, B, raccdesc, waccdesc);

            if copied != B then
                fault = TRUE;
            else
                memcpy.fromaddress = memcpy.fromaddress - B;
                memcpy.toaddress = memcpy.toaddress - B;

        if !fault then
            memcpy.cpysize = memcpy.cpysize - B;
            memcpy.stagecpysize = memcpy.stagecpysize - B;

UpdateCpyRegisters(memcpy, fault, copied);

if fault then
    if IsFault(memaddrdesc) then
        AArch64.Abort(memaddrdesc.vaddress, memaddrdesc.fault);
    else
        AccessDescriptor accdesc = if iswrite then waccdesc else raccdesc;
        HandleExternalAbort(memstatus, iswrite, memaddrdesc, B, accdesc);

if memcpy.stage == MOPSStage_Prologue then
    PSTATE.<N,Z,C,V> = memcpy.nzcv;
```

C6.2.99 CPYPTWN, CPYMTWN, CPYETWN

Memory Copy, reads and writes unprivileged, writes non-temporal. These instructions perform a memory copy. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYPTWN, then CPYMTWN, and then CPYETWN.

CPYPTWN performs some preconditioning of the arguments suitable for using the CPYMTWN instruction, and performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYMTWN performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYETWN performs the last part of the memory copy.

Note

The inclusion of IMPLEMENTATION DEFINED amounts of memory copy allows some optimization of the size that can be performed.

For CPYPTWN, the following saturation logic is applied:

If Xn<63:55> != 000000000, the copy size Xn is saturated to `0x007FFFFFFFFFFFFF`.

After that saturation logic is applied, the direction of the memory copy is based on the following algorithm:

If (Xs > Xd) && (Xd + saturated Xn) > Xs, then direction = forward

Elsif (Xs < Xd) && (Xs + saturated Xn) > Xd, then direction = backward

Else direction = IMPLEMENTATION DEFINED choice between forward and backward.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

After execution of CPYPTWN, option A (which results in encoding PSTATE.C = 0):

- PSTATE.{N,Z,V} are set to {0,0,0}.
- If the copy is in the forward direction, then:
  - Xs holds the original Xs + saturated Xn.
  - Xd holds the original Xd + saturated Xn.
  - Xn holds -1* saturated Xn + an IMPLEMENTATION DEFINED number of bytes copied.
- If the copy is in the backward direction, then:
  - Xs and Xd are unchanged.
  - Xn holds the saturated value of Xn - an IMPLEMENTATION DEFINED number of bytes copied.

After execution of CPYPTWN, option B (which results in encoding PSTATE.C = 1):

- If the copy is in the forward direction, then:
  - Xs holds the original Xs + an IMPLEMENTATION DEFINED number of bytes copied.
  - Xd holds the original Xd + an IMPLEMENTATION DEFINED number of bytes copied.
  - Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - PSTATE.{N,Z,V} are set to {0,0,0}.
- If the copy is in the backward direction, then:
  - Xs holds the original Xs + saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - Xd holds the original Xd + saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - PSTATE.{N,Z,V} are set to {1,0,0}.

For CPYMTWN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number.
- If the copy is in the forward direction (Xn is a negative number), then:
  - Xn holds -1* the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the lowest address that the copy is copied from -Xn.
  - Xd holds the lowest address that the copy is made to -Xn.
  - At the end of the instruction, the value of Xn is written back with -1* the number of bytes remaining to be copied in the memory copy in total.
- If the copy is in the backward direction (Xn is a positive number), then:
  - Xn holds the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the highest address that the copy is copied from -Xn+1.
  - Xd holds the highest address that the copy is copied to -Xn+1.
  - At the end of the instruction, the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.

For CPYMTWN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes to be copied in the memory copy in total.
- If the copy is in the forward direction (PSTATE.N == 0), then:
  - Xs holds the lowest address that the copy is copied from.
  - Xd holds the lowest address that the copy is copied to.
  - At the end of the instruction:
    - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
    - the value of Xs is written back with the lowest address that has not been copied from.
    - the value of Xd is written back with the lowest address that has not been copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
  - Xs holds the highest address that the copy is copied from +1.
  - Xd holds the highest address that the copy is copied to +1.
  - At the end of the instruction:
    - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
    - the value of Xs is written back with the highest address that has not been copied from +1.
    - the value of Xd is written back with the highest address that has not been copied to +1.

For CPYETWN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number.
- If the copy is in the forward direction (Xn is a negative number), then:
  - Xn holds -1* the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the lowest address that the copy is copied from -Xn.
  - Xd holds the lowest address that the copy is made to -Xn.
  - At the end of the instruction, the value of Xn is written back with 0.
- If the copy is in the backward direction (Xn is a positive number), then:
  - Xn holds the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the highest address that the copy is copied from -Xn+1.

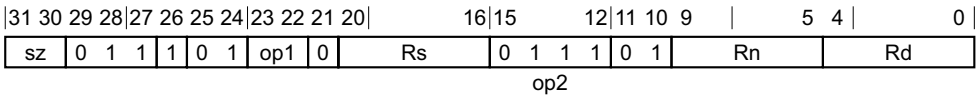
- Xd holds the highest address that the copy is copied to -Xn+1.
- At the end of the instruction, the value of Xn is written back with 0.

For CPYETWN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes to be copied in the memory copy in total.
- If the copy is in the forward direction (PSTATE.N == 0), then:
  - Xs holds the lowest address that the copy is copied from.
  - Xd holds the lowest address that the copy is copied to.
  - At the end of the instruction:
    - the value of Xn is written back with 0.
    - the value of Xs is written back with the lowest address that has not been copied from.
    - the value of Xd is written back with the lowest address that has not been copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
  - Xs holds the highest address that the copy is copied from +1.
  - Xd holds the highest address that the copy is copied to +1.
  - At the end of the instruction:
    - the value of Xn is written back with 0.
    - the value of Xs is written back with the highest address that has not been copied from +1.
    - the value of Xd is written back with the highest address that has not been copied to +1.

Integer

(FEAT_MOPS)



Epilogue variant

Applies when `op1 == 10`.

```
CPYETWN [<Xd>]!, [<Xs>]!, <Xn>!
```

Main variant

Applies when `op1 == 01`.

```
CPYMTWN [<Xd>]!, [<Xs>]!, <Xn>!
```

Prologue variant

Applies when `op1 == 00`.

```
CPYPTWN [<Xd>]!, [<Xs>]!, <Xn>!
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_MOPS) || sz != '00' then UNDEFINED;

CPYParams memcpy;
memcpy.d = UInt(Rd);
memcpy.s = UInt(Rs);
memcpy.n = UInt(Rn);
bits(4) options = op2;
```

```
boolean rnontemporal = options<3> == '1';
boolean wnontemporal = options<2> == '1';

case op1 of
    when '00' memcpy.stage = MOPSStage_Prologue;
    when '01' memcpy.stage = MOPSStage_Main;
    when '10' memcpy.stage = MOPSStage_Epilogue;
    otherwise SEE "Memory Copy and Memory Set";

CheckMOPSEnabled();

if (memcpy.s == memcpy.n || memcpy.s == memcpy.d || memcpy.n == memcpy.d || memcpy.d == 31 || memcpy.s
== 31 || memcpy.n == 31) then
    Constraint c = ConstrainUnpredictable(Unpredictable_MOPSOVERLAP31);
    assert c IN {Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNDEF UNDEFINED;
        when Constraint_NOP   EndOfInstruction();
```

Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*, and particularly *Memory Copy and Memory Set CPY**.

Assembler symbols

<Xd> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the "Rd" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.

<Xs> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the "Rs" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the "Rs" field.

<Xn> For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero at the end of the instruction, encoded in the "Rn" field.

For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the "Rn" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the "Rn" field.

Operation

```
constant integer N = MaxBlockSizeCopiedBytes();
bits(8*N) readdata;

memcpy.nzcv = PSTATE.<N,Z,C,V>;
memcpy.toaddress = X[memcpy.d, 64];
memcpy.fromaddress = X[memcpy.s, 64];

if memcpy.stage == MOPSStage_Prologue then
    memcpy.cpysize = UInt(X[memcpy.n, 64]);
else
    memcpy.cpysize = SInt(X[memcpy.n, 64]);

memcpy.implements_option_a = CPYOptionA();
```

```
boolean rprivileged = if options<1> == '1' then AArch64.IsUnprivAccessPriv() else PSTATE.EL != EL0;
boolean wprivileged = if options<0> == '1' then AArch64.IsUnprivAccessPriv() else PSTATE.EL != EL0;

AccessDescriptor raccdesc = CreateAccDescMOPS(MemOp_LOAD, rprivileged, rnontemporal);
AccessDescriptor waccdesc = CreateAccDescMOPS(MemOp_STORE, wprivileged, wnontemporal);

if memcpy.stage == MOPSStage_Prologue then
    if memcpy.cpysize > 0x007FFFFFFFFFFFF then
        memcpy.cpysize = 0x007FFFFFFFFFFFF;

    memcpy.forward = IsMemCpyForward(memcpy);

    if memcpy.implements_option_a then
        memcpy.nzcv = '0000';
        if memcpy.forward then
            // Copy in the forward direction offsets the arguments.
            memcpy.toaddress = memcpy.toaddress + memcpy.cpysize;
            memcpy.fromaddress = memcpy.fromaddress + memcpy.cpysize;
            memcpy.cpysize = 0 - memcpy.cpysize;
    else
        if !memcpy.forward then
            // Copy in the reverse direction offsets the arguments.
            memcpy.toaddress = memcpy.toaddress + memcpy.cpysize;
            memcpy.fromaddress = memcpy.fromaddress + memcpy.cpysize;
            memcpy.nzcv = '1010';
        else
            memcpy.nzcv = '0010';

memcpy.stagecpysize = MemCpyStageSize(memcpy);

if memcpy.stage != MOPSStage_Prologue then
    memcpy.forward = memcpy.cpysize < 0 || (!memcpy.implements_option_a && memcpy.nzcv<3> == '0');
    CheckMemCpyParams(memcpy, options);

integer copied;
boolean iswrite;
AddressDescriptor memaddrdesc;
PhysMemRetStatus memstatus;
boolean fault = FALSE;
integer B;

if memcpy.implements_option_a then
    while memcpy.stagecpysize != 0 && !fault do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(memcpy);

        if memcpy.forward then
            assert B <= -1 * memcpy.stagecpysize;
            (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress + memcpy.cpysize,
memcpy.fromaddress + memcpy.cpysize, memcpy.forward, B, raccdesc, waccdesc);
            if copied != B then
                fault = TRUE;
            else
                memcpy.cpysize = memcpy.cpysize + B;
                memcpy.stagecpysize = memcpy.stagecpysize + B;

        else
            assert B <= memcpy.stagecpysize;
            memcpy.cpysize = memcpy.cpysize - B;
            memcpy.stagecpysize = memcpy.stagecpysize - B;

            (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress + memcpy.cpysize,
memcpy.fromaddress + memcpy.cpysize, memcpy.forward, B, raccdesc, waccdesc);
            if copied != B then
                fault = TRUE;
                memcpy.cpysize = memcpy.cpysize + B;
                memcpy.stagecpysize = memcpy.stagecpysize + B;
```

```
else
    while memcpy.stagecpysize > 0 && !fault do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(memcpy);
        assert B <= memcpy.stagecpysize;

        if memcpy.forward then
            (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress,
memcpy.fromaddress, memcpy.forward, B, raccdesc, waccdesc);
            if copied != B then
                fault = TRUE;
            else
                memcpy.fromaddress = memcpy.fromaddress + B;
                memcpy.toaddress = memcpy.toaddress + B;
        else
            (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress - B,
memcpy.fromaddress - B, memcpy.forward, B, raccdesc, waccdesc);

            if copied != B then
                fault = TRUE;
            else
                memcpy.fromaddress = memcpy.fromaddress - B;
                memcpy.toaddress = memcpy.toaddress - B;

        if !fault then
            memcpy.cpysize = memcpy.cpysize - B;
            memcpy.stagecpysize = memcpy.stagecpysize - B;

UpdateCpyRegisters(memcpy, fault, copied);

if fault then
    if IsFault(memaddrdesc) then
        AArch64.Abort(memaddrdesc.vaddress, memaddrdesc.fault);
    else
        AccessDescriptor accdesc = if iswrite then waccdesc else raccdesc;
        HandleExternalAbort(memstatus, iswrite, memaddrdesc, B, accdesc);

if memcpy.stage == MOPSStage_Prologue then
    PSTATE.<N,Z,C,V> = memcpy.nzcv;
```

C6.2.100 CPYPWN, CPYMWN, CPYEWN

Memory Copy, writes non-temporal. These instructions perform a memory copy. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYPWN, then CPYMWN, and then CPYEWN.

CPYPWN performs some preconditioning of the arguments suitable for using the CPYMWN instruction, and performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYMWN performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYEWN performs the last part of the memory copy.

Note

The inclusion of IMPLEMENTATION DEFINED amounts of memory copy allows some optimization of the size that can be performed.

For CPYPWN, the following saturation logic is applied:

If Xn<63:55> != 000000000, the copy size Xn is saturated to `0x007FFFFFFFFFFFF`.

After that saturation logic is applied, the direction of the memory copy is based on the following algorithm:

If (Xs > Xd) && (Xd + saturated Xn) > Xs, then direction = forward

Elsif (Xs < Xd) && (Xs + saturated Xn) > Xd, then direction = backward

Else direction = IMPLEMENTATION DEFINED choice between forward and backward.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

After execution of CPYPWN, option A (which results in encoding PSTATE.C = 0):

- PSTATE.{N,Z,V} are set to {0,0,0}.
- If the copy is in the forward direction, then:
  - Xs holds the original Xs + saturated Xn.
  - Xd holds the original Xd + saturated Xn.
  - Xn holds -1* saturated Xn + an IMPLEMENTATION DEFINED number of bytes copied.
- If the copy is in the backward direction, then:
  - Xs and Xd are unchanged.
  - Xn holds the saturated value of Xn - an IMPLEMENTATION DEFINED number of bytes copied.

After execution of CPYPWN, option B (which results in encoding PSTATE.C = 1):

- If the copy is in the forward direction, then:
  - Xs holds the original Xs + an IMPLEMENTATION DEFINED number of bytes copied.
  - Xd holds the original Xd + an IMPLEMENTATION DEFINED number of bytes copied.
  - Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - PSTATE.{N,Z,V} are set to {0,0,0}.
- If the copy is in the backward direction, then:
  - Xs holds the original Xs + saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - Xd holds the original Xd + saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - PSTATE.{N,Z,V} are set to {1,0,0}.

For CPYMWN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number.
- If the copy is in the forward direction (Xn is a negative number), then:
  - Xn holds -1* the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the lowest address that the copy is copied from -Xn.
  - Xd holds the lowest address that the copy is made to -Xn.
  - At the end of the instruction, the value of Xn is written back with -1* the number of bytes remaining to be copied in the memory copy in total.
- If the copy is in the backward direction (Xn is a positive number), then:
  - Xn holds the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the highest address that the copy is copied from -Xn+1.
  - Xd holds the highest address that the copy is copied to -Xn+1.
  - At the end of the instruction, the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.

For CPYMWN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes to be copied in the memory copy in total.
- If the copy is in the forward direction (PSTATE.N == 0), then:
  - Xs holds the lowest address that the copy is copied from.
  - Xd holds the lowest address that the copy is copied to.
  - At the end of the instruction:
    - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
    - the value of Xs is written back with the lowest address that has not been copied from.
    - the value of Xd is written back with the lowest address that has not been copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
  - Xs holds the highest address that the copy is copied from +1.
  - Xd holds the highest address that the copy is copied to +1.
  - At the end of the instruction:
    - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
    - the value of Xs is written back with the highest address that has not been copied from +1.
    - the value of Xd is written back with the highest address that has not been copied to +1.

For CPYEWN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number.
- If the copy is in the forward direction (Xn is a negative number), then:
  - Xn holds -1* the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the lowest address that the copy is copied from -Xn.
  - Xd holds the lowest address that the copy is made to -Xn.
  - At the end of the instruction, the value of Xn is written back with 0.
- If the copy is in the backward direction (Xn is a positive number), then:
  - Xn holds the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the highest address that the copy is copied from -Xn+1.

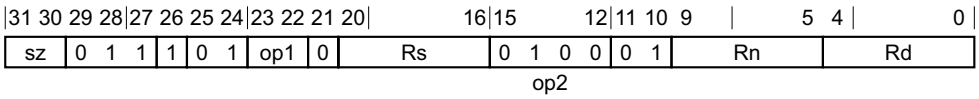
- Xd holds the highest address that the copy is copied to -Xn+1.
- At the end of the instruction, the value of Xn is written back with 0.

For CPYEWN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes to be copied in the memory copy in total.
- If the copy is in the forward direction (PSTATE.N == 0), then:
  - Xs holds the lowest address that the copy is copied from.
  - Xd holds the lowest address that the copy is copied to.
  - At the end of the instruction:
    - the value of Xn is written back with 0.
    - the value of Xs is written back with the lowest address that has not been copied from.
    - the value of Xd is written back with the lowest address that has not been copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
  - Xs holds the highest address that the copy is copied from +1.
  - Xd holds the highest address that the copy is copied to +1.
  - At the end of the instruction:
    - the value of Xn is written back with 0.
    - the value of Xs is written back with the highest address that has not been copied from +1.
    - the value of Xd is written back with the highest address that has not been copied to +1.

Integer

(FEAT_MOPS)



Epilogue variant

Applies when `op1 == 10`.

```
CPYEWN [<Xd>]!, [<Xs>]!, <Xn>!
```

Main variant

Applies when `op1 == 01`.

```
CPYMWN [<Xd>]!, [<Xs>]!, <Xn>!
```

Prologue variant

Applies when `op1 == 00`.

```
CPYPWN [<Xd>]!, [<Xs>]!, <Xn>!
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_MOPS) || sz != '00' then UNDEFINED;

CPYParams memcpy;
memcpy.d = UInt(Rd);
memcpy.s = UInt(Rs);
memcpy.n = UInt(Rn);
bits(4) options = op2;
```

```
boolean rnontemporal = options<3> == '1';
boolean wnontemporal = options<2> == '1';

case op1 of
    when '00' memcpy.stage = MOPSStage_Prologue;
    when '01' memcpy.stage = MOPSStage_Main;
    when '10' memcpy.stage = MOPSStage_Epilogue;
    otherwise SEE "Memory Copy and Memory Set";

CheckMOPSEnabled();

if (memcpy.s == memcpy.n || memcpy.s == memcpy.d || memcpy.n == memcpy.d || memcpy.d == 31 || memcpy.s
== 31 || memcpy.n == 31) then
    Constraint c = ConstrainUnpredictable(Unpredictable_MOPSOVERLAP31);
    assert c IN {Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNDEF UNDEFINED;
        when Constraint_NOP   EndOfInstruction();
```

Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*, and particularly *Memory Copy and Memory Set CPY**.

Assembler symbols

<Xd> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the "Rd" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.

<Xs> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the "Rs" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the "Rs" field.

<Xn> For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero at the end of the instruction, encoded in the "Rn" field.

For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the "Rn" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the "Rn" field.

Operation

```
constant integer N = MaxBlockSizeCopiedBytes();
bits(8*N) readdata;

memcpy.nzcv = PSTATE.<N,Z,C,V>;
memcpy.toaddress = X[memcpy.d, 64];
memcpy.fromaddress = X[memcpy.s, 64];

if memcpy.stage == MOPSStage_Prologue then
    memcpy.cpysize = UInt(X[memcpy.n, 64]);
else
    memcpy.cpysize = SInt(X[memcpy.n, 64]);

memcpy.implements_option_a = CPYOptionA();
```

```
boolean rprivileged = if options<1> == '1' then AArch64.IsUnprivAccessPriv() else PSTATE.EL != EL0;
boolean wprivileged = if options<0> == '1' then AArch64.IsUnprivAccessPriv() else PSTATE.EL != EL0;

AccessDescriptor raccdesc = CreateAccDescMOPS(MemOp_LOAD, rprivileged, rnontemporal);
AccessDescriptor waccdesc = CreateAccDescMOPS(MemOp_STORE, wprivileged, wnontemporal);

if memcpy.stage == MOPSStage_Prologue then
    if memcpy.cpysize > 0x007FFFFFFFFFFFF then
        memcpy.cpysize = 0x007FFFFFFFFFFFF;

    memcpy.forward = IsMemCpyForward(memcpy);

    if memcpy.implements_option_a then
        memcpy.nzcv = '0000';
        if memcpy.forward then
            // Copy in the forward direction offsets the arguments.
            memcpy.toaddress = memcpy.toaddress + memcpy.cpysize;
            memcpy.fromaddress = memcpy.fromaddress + memcpy.cpysize;
            memcpy.cpysize = 0 - memcpy.cpysize;
    else
        if !memcpy.forward then
            // Copy in the reverse direction offsets the arguments.
            memcpy.toaddress = memcpy.toaddress + memcpy.cpysize;
            memcpy.fromaddress = memcpy.fromaddress + memcpy.cpysize;
            memcpy.nzcv = '1010';
        else
            memcpy.nzcv = '0010';

memcpy.stagecpysize = MemCpyStageSize(memcpy);

if memcpy.stage != MOPSStage_Prologue then
    memcpy.forward = memcpy.cpysize < 0 || (!memcpy.implements_option_a && memcpy.nzcv<3> == '0');
    CheckMemCpyParams(memcpy, options);

integer copied;
boolean iswrite;
AddressDescriptor memaddrdesc;
PhysMemRetStatus memstatus;
boolean fault = FALSE;
integer B;

if memcpy.implements_option_a then
    while memcpy.stagecpysize != 0 && !fault do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(memcpy);

        if memcpy.forward then
            assert B <= -1 * memcpy.stagecpysize;
            (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress + memcpy.cpysize,
memcpy.fromaddress + memcpy.cpysize, memcpy.forward, B, raccdesc, waccdesc);
            if copied != B then
                fault = TRUE;
            else
                memcpy.cpysize = memcpy.cpysize + B;
                memcpy.stagecpysize = memcpy.stagecpysize + B;

        else
            assert B <= memcpy.stagecpysize;
            memcpy.cpysize = memcpy.cpysize - B;
            memcpy.stagecpysize = memcpy.stagecpysize - B;

            (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress + memcpy.cpysize,
memcpy.fromaddress + memcpy.cpysize, memcpy.forward, B, raccdesc, waccdesc);
            if copied != B then
                fault = TRUE;
                memcpy.cpysize = memcpy.cpysize + B;
                memcpy.stagecpysize = memcpy.stagecpysize + B;
```

```
else
    while memcpy.stagecpysize > 0 && !fault do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(memcpy);
        assert B <= memcpy.stagecpysize;

        if memcpy.forward then
            (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress,
memcpy.fromaddress, memcpy.forward, B, raccdesc, waccdesc);
            if copied != B then
                fault = TRUE;
            else
                memcpy.fromaddress = memcpy.fromaddress + B;
                memcpy.toaddress = memcpy.toaddress + B;
        else
            (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress - B,
memcpy.fromaddress - B, memcpy.forward, B, raccdesc, waccdesc);

            if copied != B then
                fault = TRUE;
            else
                memcpy.fromaddress = memcpy.fromaddress - B;
                memcpy.toaddress = memcpy.toaddress - B;

        if !fault then
            memcpy.cpysize = memcpy.cpysize - B;
            memcpy.stagecpysize = memcpy.stagecpysize - B;

UpdateCpyRegisters(memcpy, fault, copied);

if fault then
    if IsFault(memaddrdesc) then
        AArch64.Abort(memaddrdesc.vaddress, memaddrdesc.fault);
    else
        AccessDescriptor accdesc = if iswrite then waccdesc else raccdesc;
        HandleExternalAbort(memstatus, iswrite, memaddrdesc, B, accdesc);

if memcpy.stage == MOPSStage_Prologue then
    PSTATE.<N,Z,C,V> = memcpy.nzcv;
```

C6.2.101 CPYPWT, CPYMWT, CPYEWT

Memory Copy, writes unprivileged. These instructions perform a memory copy. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYPWT, then CPYMWT, and then CPYEWT.

CPYPWT performs some preconditioning of the arguments suitable for using the CPYMWT instruction, and performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYMWT performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYEWT performs the last part of the memory copy.

Note

The inclusion of IMPLEMENTATION DEFINED amounts of memory copy allows some optimization of the size that can be performed.

For CPYPWT, the following saturation logic is applied:

If Xn<63:55> != 000000000, the copy size Xn is saturated to `0x007FFFFFFFFFFFFF`.

After that saturation logic is applied, the direction of the memory copy is based on the following algorithm:

If (Xs > Xd) && (Xd + saturated Xn) > Xs, then direction = forward

Elsif (Xs < Xd) && (Xs + saturated Xn) > Xd, then direction = backward

Else direction = IMPLEMENTATION DEFINED choice between forward and backward.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

After execution of CPYPWT, option A (which results in encoding PSTATE.C = 0):

- PSTATE.{N,Z,V} are set to {0,0,0}.
- If the copy is in the forward direction, then:
  - Xs holds the original Xs + saturated Xn.
  - Xd holds the original Xd + saturated Xn.
  - Xn holds -1* saturated Xn + an IMPLEMENTATION DEFINED number of bytes copied.
- If the copy is in the backward direction, then:
  - Xs and Xd are unchanged.
  - Xn holds the saturated value of Xn - an IMPLEMENTATION DEFINED number of bytes copied.

After execution of CPYPWT, option B (which results in encoding PSTATE.C = 1):

- If the copy is in the forward direction, then:
  - Xs holds the original Xs + an IMPLEMENTATION DEFINED number of bytes copied.
  - Xd holds the original Xd + an IMPLEMENTATION DEFINED number of bytes copied.
  - Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - PSTATE.{N,Z,V} are set to {0,0,0}.
- If the copy is in the backward direction, then:
  - Xs holds the original Xs + saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - Xd holds the original Xd + saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - PSTATE.{N,Z,V} are set to {1,0,0}.

For CPYMWT, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number.
- If the copy is in the forward direction (Xn is a negative number), then:
  - Xn holds -1* the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the lowest address that the copy is copied from -Xn.
  - Xd holds the lowest address that the copy is made to -Xn.
  - At the end of the instruction, the value of Xn is written back with -1* the number of bytes remaining to be copied in the memory copy in total.
- If the copy is in the backward direction (Xn is a positive number), then:
  - Xn holds the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the highest address that the copy is copied from -Xn+1.
  - Xd holds the highest address that the copy is copied to -Xn+1.
  - At the end of the instruction, the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.

For CPYMWT, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes to be copied in the memory copy in total.
- If the copy is in the forward direction (PSTATE.N == 0), then:
  - Xs holds the lowest address that the copy is copied from.
  - Xd holds the lowest address that the copy is copied to.
  - At the end of the instruction:
    - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
    - the value of Xs is written back with the lowest address that has not been copied from.
    - the value of Xd is written back with the lowest address that has not been copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
  - Xs holds the highest address that the copy is copied from +1.
  - Xd holds the highest address that the copy is copied to +1.
  - At the end of the instruction:
    - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
    - the value of Xs is written back with the highest address that has not been copied from +1.
    - the value of Xd is written back with the highest address that has not been copied to +1.

For CPYEWT, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number.
- If the copy is in the forward direction (Xn is a negative number), then:
  - Xn holds -1* the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the lowest address that the copy is copied from -Xn.
  - Xd holds the lowest address that the copy is made to -Xn.
  - At the end of the instruction, the value of Xn is written back with 0.
- If the copy is in the backward direction (Xn is a positive number), then:
  - Xn holds the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the highest address that the copy is copied from -Xn+1.

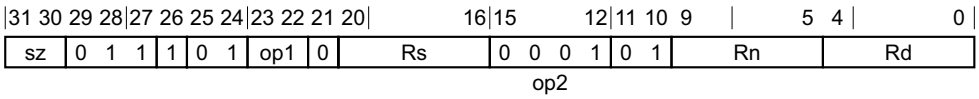
  - Xd holds the highest address that the copy is copied to -Xn+1.
  - At the end of the instruction, the value of Xn is written back with 0.

For CPYEWT, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes to be copied in the memory copy in total.
- If the copy is in the forward direction (PSTATE.N == 0), then:
  - Xs holds the lowest address that the copy is copied from.
  - Xd holds the lowest address that the copy is copied to.
  - At the end of the instruction:
    - the value of Xn is written back with 0.
    - the value of Xs is written back with the lowest address that has not been copied from.
    - the value of Xd is written back with the lowest address that has not been copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
  - Xs holds the highest address that the copy is copied from +1.
  - Xd holds the highest address that the copy is copied to +1.
  - At the end of the instruction:
    - the value of Xn is written back with 0.
    - the value of Xs is written back with the highest address that has not been copied from +1.
    - the value of Xd is written back with the highest address that has not been copied to +1.

Integer

(FEAT_MOPS)



Epilogue variant

Applies when `op1 == 10`.

```
CPYEWT [<Xd>]!, [<Xs>]!, <Xn>!
```

Main variant

Applies when `op1 == 01`.

```
CPYMWT [<Xd>]!, [<Xs>]!, <Xn>!
```

Prologue variant

Applies when `op1 == 00`.

```
CPYPWT [<Xd>]!, [<Xs>]!, <Xn>!
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_MOPS) || sz != '00' then UNDEFINED;

CPYParams memcpy;
memcpy.d = UInt(Rd);
memcpy.s = UInt(Rs);
memcpy.n = UInt(Rn);
bits(4) options = op2;
```

```
boolean rnontemporal = options<3> == '1';
boolean wnontemporal = options<2> == '1';

case op1 of
    when '00' memcpy.stage = MOPSStage_Prologue;
    when '01' memcpy.stage = MOPSStage_Main;
    when '10' memcpy.stage = MOPSStage_Epilogue;
    otherwise SEE "Memory Copy and Memory Set";

CheckMOPSEnabled();

if (memcpy.s == memcpy.n || memcpy.s == memcpy.d || memcpy.n == memcpy.d || memcpy.d == 31 || memcpy.s
== 31 || memcpy.n == 31) then
    Constraint c = ConstrainUnpredictable(Unpredictable_MOPSOVERLAP31);
    assert c IN {Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNDEF UNDEFINED;
        when Constraint_NOP   EndOfInstruction();
```

Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*, and particularly *Memory Copy and Memory Set CPY**.

Assembler symbols

<Xd> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the "Rd" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.

<Xs> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the "Rs" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the "Rs" field.

<Xn> For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero at the end of the instruction, encoded in the "Rn" field.

For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the "Rn" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the "Rn" field.

Operation

```
constant integer N = MaxBlockSizeCopiedBytes();
bits(8*N) readdata;

memcpy.nzcv = PSTATE.<N,Z,C,V>;
memcpy.toaddress = X[memcpy.d, 64];
memcpy.fromaddress = X[memcpy.s, 64];

if memcpy.stage == MOPSStage_Prologue then
    memcpy.cpysize = UInt(X[memcpy.n, 64]);
else
    memcpy.cpysize = SInt(X[memcpy.n, 64]);

memcpy.implements_option_a = CPYOptionA();
```

```
boolean rprivileged = if options<1> == '1' then AArch64.IsUnprivAccessPriv() else PSTATE.EL != EL0;
boolean wprivileged = if options<0> == '1' then AArch64.IsUnprivAccessPriv() else PSTATE.EL != EL0;

AccessDescriptor raccdesc = CreateAccDescMOPS(MemOp_LOAD, rprivileged, rnontemporal);
AccessDescriptor waccdesc = CreateAccDescMOPS(MemOp_STORE, wprivileged, wnontemporal);

if memcpy.stage == MOPSStage_Prologue then
    if memcpy.cpysize > 0x007FFFFFFFFFFFF then
        memcpy.cpysize = 0x007FFFFFFFFFFFF;

    memcpy.forward = IsMemCpyForward(memcpy);

    if memcpy.implements_option_a then
        memcpy.nzcv = '0000';
        if memcpy.forward then
            // Copy in the forward direction offsets the arguments.
            memcpy.toaddress = memcpy.toaddress + memcpy.cpysize;
            memcpy.fromaddress = memcpy.fromaddress + memcpy.cpysize;
            memcpy.cpysize = 0 - memcpy.cpysize;
    else
        if !memcpy.forward then
            // Copy in the reverse direction offsets the arguments.
            memcpy.toaddress = memcpy.toaddress + memcpy.cpysize;
            memcpy.fromaddress = memcpy.fromaddress + memcpy.cpysize;
            memcpy.nzcv = '1010';
        else
            memcpy.nzcv = '0010';

memcpy.stagecpysize = MemCpyStageSize(memcpy);

if memcpy.stage != MOPSStage_Prologue then
    memcpy.forward = memcpy.cpysize < 0 || (!memcpy.implements_option_a && memcpy.nzcv<3> == '0');
    CheckMemCpyParams(memcpy, options);

integer copied;
boolean iswrite;
AddressDescriptor memaddrdesc;
PhysMemRetStatus memstatus;
boolean fault = FALSE;
integer B;

if memcpy.implements_option_a then
    while memcpy.stagecpysize != 0 && !fault do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(memcpy);

        if memcpy.forward then
            assert B <= -1 * memcpy.stagecpysize;
            (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress + memcpy.cpysize,
memcpy.fromaddress + memcpy.cpysize, memcpy.forward, B, raccdesc, waccdesc);
            if copied != B then
                fault = TRUE;
            else
                memcpy.cpysize = memcpy.cpysize + B;
                memcpy.stagecpysize = memcpy.stagecpysize + B;

        else
            assert B <= memcpy.stagecpysize;
            memcpy.cpysize = memcpy.cpysize - B;
            memcpy.stagecpysize = memcpy.stagecpysize - B;

            (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress + memcpy.cpysize,
memcpy.fromaddress + memcpy.cpysize, memcpy.forward, B, raccdesc, waccdesc);
            if copied != B then
                fault = TRUE;
                memcpy.cpysize = memcpy.cpysize + B;
                memcpy.stagecpysize = memcpy.stagecpysize + B;
```

```
else
    while memcpy.stagecpysize > 0 && !fault do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(memcpy);
        assert B <= memcpy.stagecpysize;

        if memcpy.forward then
            (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress,
memcpy.fromaddress, memcpy.forward, B, raccdesc, waccdesc);
            if copied != B then
                fault = TRUE;
            else
                memcpy.fromaddress = memcpy.fromaddress + B;
                memcpy.toaddress = memcpy.toaddress + B;
        else
            (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress - B,
memcpy.fromaddress - B, memcpy.forward, B, raccdesc, waccdesc);

            if copied != B then
                fault = TRUE;
            else
                memcpy.fromaddress = memcpy.fromaddress - B;
                memcpy.toaddress = memcpy.toaddress - B;

        if !fault then
            memcpy.cpysize = memcpy.cpysize - B;
            memcpy.stagecpysize = memcpy.stagecpysize - B;

UpdateCpyRegisters(memcpy, fault, copied);

if fault then
    if IsFault(memaddrdesc) then
        AArch64.Abort(memaddrdesc.vaddress, memaddrdesc.fault);
    else
        AccessDescriptor accdesc = if iswrite then waccdesc else raccdesc;
        HandleExternalAbort(memstatus, iswrite, memaddrdesc, B, accdesc);

if memcpy.stage == MOPSStage_Prologue then
    PSTATE.<N,Z,C,V> = memcpy.nzcv;
```

C6.2.102 CPYPWTN, CPYMWTN, CPYEWTN

Memory Copy, writes unprivileged, reads and writes non-temporal. These instructions perform a memory copy. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYPWTN, then CPYMWTN, and then CPYEWTN.

CPYPWTN performs some preconditioning of the arguments suitable for using the CPYMWTN instruction, and performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYMWTN performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYEWTN performs the last part of the memory copy.

Note

The inclusion of IMPLEMENTATION DEFINED amounts of memory copy allows some optimization of the size that can be performed.

For CPYPWTN, the following saturation logic is applied:

If Xn<63:55> != 000000000, the copy size Xn is saturated to `0x007FFFFFFFFFFFF`.

After that saturation logic is applied, the direction of the memory copy is based on the following algorithm:

If (Xs > Xd) && (Xd + saturated Xn) > Xs, then direction = forward

Elsif (Xs < Xd) && (Xs + saturated Xn) > Xd, then direction = backward

Else direction = IMPLEMENTATION DEFINED choice between forward and backward.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

After execution of CPYPWTN, option A (which results in encoding PSTATE.C = 0):

- PSTATE.{N,Z,V} are set to {0,0,0}.
- If the copy is in the forward direction, then:
  - Xs holds the original Xs + saturated Xn.
  - Xd holds the original Xd + saturated Xn.
  - Xn holds -1* saturated Xn + an IMPLEMENTATION DEFINED number of bytes copied.
- If the copy is in the backward direction, then:
  - Xs and Xd are unchanged.
  - Xn holds the saturated value of Xn - an IMPLEMENTATION DEFINED number of bytes copied.

After execution of CPYPWTN, option B (which results in encoding PSTATE.C = 1):

- If the copy is in the forward direction, then:
  - Xs holds the original Xs + an IMPLEMENTATION DEFINED number of bytes copied.
  - Xd holds the original Xd + an IMPLEMENTATION DEFINED number of bytes copied.
  - Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - PSTATE.{N,Z,V} are set to {0,0,0}.
- If the copy is in the backward direction, then:
  - Xs holds the original Xs + saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - Xd holds the original Xd + saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - PSTATE.{N,Z,V} are set to {1,0,0}.

For CPYMWTN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number.
- If the copy is in the forward direction (Xn is a negative number), then:
  - Xn holds -1* the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the lowest address that the copy is copied from -Xn.
  - Xd holds the lowest address that the copy is made to -Xn.
  - At the end of the instruction, the value of Xn is written back with -1* the number of bytes remaining to be copied in the memory copy in total.
- If the copy is in the backward direction (Xn is a positive number), then:
  - Xn holds the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the highest address that the copy is copied from -Xn+1.
  - Xd holds the highest address that the copy is copied to -Xn+1.
  - At the end of the instruction, the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.

For CPYMWTN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes to be copied in the memory copy in total.
- If the copy is in the forward direction (PSTATE.N == 0), then:
  - Xs holds the lowest address that the copy is copied from.
  - Xd holds the lowest address that the copy is copied to.
  - At the end of the instruction:
    - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
    - the value of Xs is written back with the lowest address that has not been copied from.
    - the value of Xd is written back with the lowest address that has not been copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
  - Xs holds the highest address that the copy is copied from +1.
  - Xd holds the highest address that the copy is copied to +1.
  - At the end of the instruction:
    - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
    - the value of Xs is written back with the highest address that has not been copied from +1.
    - the value of Xd is written back with the highest address that has not been copied to +1.

For CPYEWTN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number.
- If the copy is in the forward direction (Xn is a negative number), then:
  - Xn holds -1* the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the lowest address that the copy is copied from -Xn.
  - Xd holds the lowest address that the copy is made to -Xn.
  - At the end of the instruction, the value of Xn is written back with 0.
- If the copy is in the backward direction (Xn is a positive number), then:
  - Xn holds the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the highest address that the copy is copied from -Xn+1.

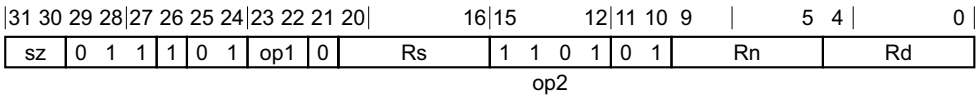
- Xd holds the highest address that the copy is copied to -Xn+1.
- At the end of the instruction, the value of Xn is written back with 0.

For CPYEWTN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes to be copied in the memory copy in total.
- If the copy is in the forward direction (PSTATE.N == 0), then:
  - Xs holds the lowest address that the copy is copied from.
  - Xd holds the lowest address that the copy is copied to.
  - At the end of the instruction:
    - the value of Xn is written back with 0.
    - the value of Xs is written back with the lowest address that has not been copied from.
    - the value of Xd is written back with the lowest address that has not been copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
  - Xs holds the highest address that the copy is copied from +1.
  - Xd holds the highest address that the copy is copied to +1.
  - At the end of the instruction:
    - the value of Xn is written back with 0.
    - the value of Xs is written back with the highest address that has not been copied from +1.
    - the value of Xd is written back with the highest address that has not been copied to +1.

Integer

(FEAT_MOPS)



Epilogue variant

Applies when `op1 == 10`.

```
CPYEWTN [<Xd>]!, [<Xs>]!, <Xn>!
```

Main variant

Applies when `op1 == 01`.

```
CPYMWTN [<Xd>]!, [<Xs>]!, <Xn>!
```

Prologue variant

Applies when `op1 == 00`.

```
CPYPWTN [<Xd>]!, [<Xs>]!, <Xn>!
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_MOPS) || sz != '00' then UNDEFINED;

CPYParams memcpy;
memcpy.d = UInt(Rd);
memcpy.s = UInt(Rs);
memcpy.n = UInt(Rn);
bits(4) options = op2;
```

```
boolean rnontemporal = options<3> == '1';
boolean wnontemporal = options<2> == '1';

case op1 of
    when '00' memcpy.stage = MOPSStage_Prologue;
    when '01' memcpy.stage = MOPSStage_Main;
    when '10' memcpy.stage = MOPSStage_Epilogue;
    otherwise SEE "Memory Copy and Memory Set";

CheckMOPSEnabled();

if (memcpy.s == memcpy.n || memcpy.s == memcpy.d || memcpy.n == memcpy.d || memcpy.d == 31 || memcpy.s
== 31 || memcpy.n == 31) then
    Constraint c = ConstrainUnpredictable(Unpredictable_MOPSOVERLAP31);
    assert c IN {Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNDEF UNDEFINED;
        when Constraint_NOP   EndOfInstruction();
```

Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*, and particularly *Memory Copy and Memory Set CPY**.

Assembler symbols

<Xd> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the "Rd" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.

<Xs> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the "Rs" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the "Rs" field.

<Xn> For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero at the end of the instruction, encoded in the "Rn" field.

For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the "Rn" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the "Rn" field.

Operation

```
constant integer N = MaxBlockSizeCopiedBytes();
bits(8*N) readdata;

memcpy.nzcv = PSTATE.<N,Z,C,V>;
memcpy.toaddress = X[memcpy.d, 64];
memcpy.fromaddress = X[memcpy.s, 64];

if memcpy.stage == MOPSStage_Prologue then
    memcpy.cpysize = UInt(X[memcpy.n, 64]);
else
    memcpy.cpysize = SInt(X[memcpy.n, 64]);

memcpy.implements_option_a = CPYOptionA();
```

```
boolean rprivileged = if options<1> == '1' then AArch64.IsUnprivAccessPriv() else PSTATE.EL != EL0;
boolean wprivileged = if options<0> == '1' then AArch64.IsUnprivAccessPriv() else PSTATE.EL != EL0;

AccessDescriptor raccdesc = CreateAccDescMOPS(MemOp_LOAD, rprivileged, rnontemporal);
AccessDescriptor waccdesc = CreateAccDescMOPS(MemOp_STORE, wprivileged, wnontemporal);

if memcpy.stage == MOPSStage_Prologue then
    if memcpy.cpysize > 0x007FFFFFFFFFFFF then
        memcpy.cpysize = 0x007FFFFFFFFFFFF;

    memcpy.forward = IsMemCpyForward(memcpy);

    if memcpy.implements_option_a then
        memcpy.nzcv = '0000';
        if memcpy.forward then
            // Copy in the forward direction offsets the arguments.
            memcpy.toaddress = memcpy.toaddress + memcpy.cpysize;
            memcpy.fromaddress = memcpy.fromaddress + memcpy.cpysize;
            memcpy.cpysize = 0 - memcpy.cpysize;
    else
        if !memcpy.forward then
            // Copy in the reverse direction offsets the arguments.
            memcpy.toaddress = memcpy.toaddress + memcpy.cpysize;
            memcpy.fromaddress = memcpy.fromaddress + memcpy.cpysize;
            memcpy.nzcv = '1010';
        else
            memcpy.nzcv = '0010';

memcpy.stagecpysize = MemCpyStageSize(memcpy);

if memcpy.stage != MOPSStage_Prologue then
    memcpy.forward = memcpy.cpysize < 0 || (!memcpy.implements_option_a && memcpy.nzcv<3> == '0');
    CheckMemCpyParams(memcpy, options);

integer copied;
boolean iswrite;
AddressDescriptor memaddrdesc;
PhysMemRetStatus memstatus;
boolean fault = FALSE;
integer B;

if memcpy.implements_option_a then
    while memcpy.stagecpysize != 0 && !fault do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(memcpy);

        if memcpy.forward then
            assert B <= -1 * memcpy.stagecpysize;
            (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress + memcpy.cpysize,
memcpy.fromaddress + memcpy.cpysize, memcpy.forward, B, raccdesc, waccdesc);
            if copied != B then
                fault = TRUE;
            else
                memcpy.cpysize = memcpy.cpysize + B;
                memcpy.stagecpysize = memcpy.stagecpysize + B;

        else
            assert B <= memcpy.stagecpysize;
            memcpy.cpysize = memcpy.cpysize - B;
            memcpy.stagecpysize = memcpy.stagecpysize - B;

            (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress + memcpy.cpysize,
memcpy.fromaddress + memcpy.cpysize, memcpy.forward, B, raccdesc, waccdesc);
            if copied != B then
                fault = TRUE;
                memcpy.cpysize = memcpy.cpysize + B;
                memcpy.stagecpysize = memcpy.stagecpysize + B;
```

```
else
    while memcpy.stagecpysize > 0 && !fault do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(memcpy);
        assert B <= memcpy.stagecpysize;

        if memcpy.forward then
            (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress,
memcpy.fromaddress, memcpy.forward, B, raccdesc, waccdesc);
            if copied != B then
                fault = TRUE;
            else
                memcpy.fromaddress = memcpy.fromaddress + B;
                memcpy.toaddress = memcpy.toaddress + B;
        else
            (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress - B,
memcpy.fromaddress - B, memcpy.forward, B, raccdesc, waccdesc);

            if copied != B then
                fault = TRUE;
            else
                memcpy.fromaddress = memcpy.fromaddress - B;
                memcpy.toaddress = memcpy.toaddress - B;

        if !fault then
            memcpy.cpysize = memcpy.cpysize - B;
            memcpy.stagecpysize = memcpy.stagecpysize - B;

UpdateCpyRegisters(memcpy, fault, copied);

if fault then
    if IsFault(memaddrdesc) then
        AArch64.Abort(memaddrdesc.vaddress, memaddrdesc.fault);
    else
        AccessDescriptor accdesc = if iswrite then waccdesc else raccdesc;
        HandleExternalAbort(memstatus, iswrite, memaddrdesc, B, accdesc);

if memcpy.stage == MOPSStage_Prologue then
    PSTATE.<N,Z,C,V> = memcpy.nzcv;
```

C6.2.103 CPYPWTRN, CPYMWTRN, CPYEWTRN

Memory Copy, writes unprivileged, reads non-temporal. These instructions perform a memory copy. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYPWTRN, then CPYMWTRN, and then CPYEWTRN.

CPYPWTRN performs some preconditioning of the arguments suitable for using the CPYMWTRN instruction, and performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYMWTRN performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYEWTRN performs the last part of the memory copy.

Note

The inclusion of IMPLEMENTATION DEFINED amounts of memory copy allows some optimization of the size that can be performed.

For CPYPWTRN, the following saturation logic is applied:

If Xn<63:55> != 000000000, the copy size Xn is saturated to `0x007FFFFFFFFFFFFF`.

After that saturation logic is applied, the direction of the memory copy is based on the following algorithm:

If (Xs > Xd) && (Xd + saturated Xn) > Xs, then direction = forward

Elsif (Xs < Xd) && (Xs + saturated Xn) > Xd, then direction = backward

Else direction = IMPLEMENTATION DEFINED choice between forward and backward.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

After execution of CPYPWTRN, option A (which results in encoding PSTATE.C = 0):

- PSTATE.{N,Z,V} are set to {0,0,0}.
- If the copy is in the forward direction, then:
  - Xs holds the original Xs + saturated Xn.
  - Xd holds the original Xd + saturated Xn.
  - Xn holds -1* saturated Xn + an IMPLEMENTATION DEFINED number of bytes copied.
- If the copy is in the backward direction, then:
  - Xs and Xd are unchanged.
  - Xn holds the saturated value of Xn - an IMPLEMENTATION DEFINED number of bytes copied.

After execution of CPYPWTRN, option B (which results in encoding PSTATE.C = 1):

- If the copy is in the forward direction, then:
  - Xs holds the original Xs + an IMPLEMENTATION DEFINED number of bytes copied.
  - Xd holds the original Xd + an IMPLEMENTATION DEFINED number of bytes copied.
  - Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - PSTATE.{N,Z,V} are set to {0,0,0}.
- If the copy is in the backward direction, then:
  - Xs holds the original Xs + saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - Xd holds the original Xd + saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - PSTATE.{N,Z,V} are set to {1,0,0}.

For CPYMWTRN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number.
- If the copy is in the forward direction (Xn is a negative number), then:
  - Xn holds -1* the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the lowest address that the copy is copied from -Xn.
  - Xd holds the lowest address that the copy is made to -Xn.
  - At the end of the instruction, the value of Xn is written back with -1* the number of bytes remaining to be copied in the memory copy in total.
- If the copy is in the backward direction (Xn is a positive number), then:
  - Xn holds the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the highest address that the copy is copied from -Xn+1.
  - Xd holds the highest address that the copy is copied to -Xn+1.
  - At the end of the instruction, the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.

For CPYMWTRN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes to be copied in the memory copy in total.
- If the copy is in the forward direction (PSTATE.N == 0), then:
  - Xs holds the lowest address that the copy is copied from.
  - Xd holds the lowest address that the copy is copied to.
  - At the end of the instruction:
    - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
    - the value of Xs is written back with the lowest address that has not been copied from.
    - the value of Xd is written back with the lowest address that has not been copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
  - Xs holds the highest address that the copy is copied from +1.
  - Xd holds the highest address that the copy is copied to +1.
  - At the end of the instruction:
    - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
    - the value of Xs is written back with the highest address that has not been copied from +1.
    - the value of Xd is written back with the highest address that has not been copied to +1.

For CPYEWTRN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number.
- If the copy is in the forward direction (Xn is a negative number), then:
  - Xn holds -1* the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the lowest address that the copy is copied from -Xn.
  - Xd holds the lowest address that the copy is made to -Xn.
  - At the end of the instruction, the value of Xn is written back with 0.
- If the copy is in the backward direction (Xn is a positive number), then:
  - Xn holds the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the highest address that the copy is copied from -Xn+1.

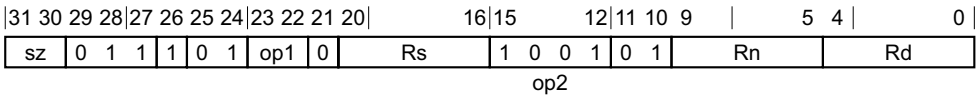
  - Xd holds the highest address that the copy is copied to -Xn+1.
  - At the end of the instruction, the value of Xn is written back with 0.

For CPYEWTRN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes to be copied in the memory copy in total.
- If the copy is in the forward direction (PSTATE.N == 0), then:
  - Xs holds the lowest address that the copy is copied from.
  - Xd holds the lowest address that the copy is copied to.
  - At the end of the instruction:
    - the value of Xn is written back with 0.
    - the value of Xs is written back with the lowest address that has not been copied from.
    - the value of Xd is written back with the lowest address that has not been copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
  - Xs holds the highest address that the copy is copied from +1.
  - Xd holds the highest address that the copy is copied to +1.
  - At the end of the instruction:
    - the value of Xn is written back with 0.
    - the value of Xs is written back with the highest address that has not been copied from +1.
    - the value of Xd is written back with the highest address that has not been copied to +1.

Integer

(FEAT_MOPS)



Epilogue variant

Applies when `op1 == 10`.

```
CPYEWTRN [<Xd>]!, [<Xs>]!, <Xn>!
```

Main variant

Applies when `op1 == 01`.

```
CPYMWTRN [<Xd>]!, [<Xs>]!, <Xn>!
```

Prologue variant

Applies when `op1 == 00`.

```
CPYPWTRN [<Xd>]!, [<Xs>]!, <Xn>!
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_MOPS) || sz != '00' then UNDEFINED;

CPYParams memcpy;
memcpy.d = UInt(Rd);
memcpy.s = UInt(Rs);
memcpy.n = UInt(Rn);
bits(4) options = op2;
```

```
boolean rnontemporal = options<3> == '1';
boolean wnontemporal = options<2> == '1';

case op1 of
    when '00' memcpy.stage = MOPSStage_Prologue;
    when '01' memcpy.stage = MOPSStage_Main;
    when '10' memcpy.stage = MOPSStage_Epilogue;
    otherwise SEE "Memory Copy and Memory Set";

CheckMOPSEnabled();

if (memcpy.s == memcpy.n || memcpy.s == memcpy.d || memcpy.n == memcpy.d || memcpy.d == 31 || memcpy.s
== 31 || memcpy.n == 31) then
    Constraint c = ConstrainUnpredictable(Unpredictable_MOPSOVERLAP31);
    assert c IN {Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNDEF UNDEFINED;
        when Constraint_NOP   EndOfInstruction();
```

Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*, and particularly *Memory Copy and Memory Set CPY**.

Assembler symbols

<Xd> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the "Rd" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.

<Xs> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the "Rs" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the "Rs" field.

<Xn> For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero at the end of the instruction, encoded in the "Rn" field.

For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the "Rn" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the "Rn" field.

Operation

```
constant integer N = MaxBlockSizeCopiedBytes();
bits(8*N) readdata;

memcpy.nzcv = PSTATE.<N,Z,C,V>;
memcpy.toaddress = X[memcpy.d, 64];
memcpy.fromaddress = X[memcpy.s, 64];

if memcpy.stage == MOPSStage_Prologue then
    memcpy.cpysize = UInt(X[memcpy.n, 64]);
else
    memcpy.cpysize = SInt(X[memcpy.n, 64]);

memcpy.implements_option_a = CPYOptionA();
```

```
boolean rprivileged = if options<1> == '1' then AArch64.IsUnprivAccessPriv() else PSTATE.EL != EL0;
boolean wprivileged = if options<0> == '1' then AArch64.IsUnprivAccessPriv() else PSTATE.EL != EL0;

AccessDescriptor raccdesc = CreateAccDescMOPS(MemOp_LOAD, rprivileged, rnontemporal);
AccessDescriptor waccdesc = CreateAccDescMOPS(MemOp_STORE, wprivileged, wnontemporal);

if memcpy.stage == MOPSStage_Prologue then
    if memcpy.cpysize > 0x007FFFFFFFFFFFF then
        memcpy.cpysize = 0x007FFFFFFFFFFFF;

    memcpy.forward = IsMemCpyForward(memcpy);

    if memcpy.implements_option_a then
        memcpy.nzcv = '0000';
        if memcpy.forward then
            // Copy in the forward direction offsets the arguments.
            memcpy.toaddress = memcpy.toaddress + memcpy.cpysize;
            memcpy.fromaddress = memcpy.fromaddress + memcpy.cpysize;
            memcpy.cpysize = 0 - memcpy.cpysize;
    else
        if !memcpy.forward then
            // Copy in the reverse direction offsets the arguments.
            memcpy.toaddress = memcpy.toaddress + memcpy.cpysize;
            memcpy.fromaddress = memcpy.fromaddress + memcpy.cpysize;
            memcpy.nzcv = '1010';
        else
            memcpy.nzcv = '0010';

memcpy.stagecpysize = MemCpyStageSize(memcpy);

if memcpy.stage != MOPSStage_Prologue then
    memcpy.forward = memcpy.cpysize < 0 || (!memcpy.implements_option_a && memcpy.nzcv<3> == '0');
    CheckMemCpyParams(memcpy, options);

integer copied;
boolean iswrite;
AddressDescriptor memaddrdesc;
PhysMemRetStatus memstatus;
boolean fault = FALSE;
integer B;

if memcpy.implements_option_a then
    while memcpy.stagecpysize != 0 && !fault do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(memcpy);

        if memcpy.forward then
            assert B <= -1 * memcpy.stagecpysize;
            (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress + memcpy.cpysize,
memcpy.fromaddress + memcpy.cpysize, memcpy.forward, B, raccdesc, waccdesc);
            if copied != B then
                fault = TRUE;
            else
                memcpy.cpysize = memcpy.cpysize + B;
                memcpy.stagecpysize = memcpy.stagecpysize + B;

        else
            assert B <= memcpy.stagecpysize;
            memcpy.cpysize = memcpy.cpysize - B;
            memcpy.stagecpysize = memcpy.stagecpysize - B;

            (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress + memcpy.cpysize,
memcpy.fromaddress + memcpy.cpysize, memcpy.forward, B, raccdesc, waccdesc);
            if copied != B then
                fault = TRUE;
                memcpy.cpysize = memcpy.cpysize + B;
                memcpy.stagecpysize = memcpy.stagecpysize + B;
```

```
else
    while memcpy.stagecpysize > 0 && !fault do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(memcpy);
        assert B <= memcpy.stagecpysize;

        if memcpy.forward then
            (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress,
memcpy.fromaddress, memcpy.forward, B, raccdesc, waccdesc);
            if copied != B then
                fault = TRUE;
            else
                memcpy.fromaddress = memcpy.fromaddress + B;
                memcpy.toaddress = memcpy.toaddress + B;
        else
            (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress - B,
memcpy.fromaddress - B, memcpy.forward, B, raccdesc, waccdesc);

            if copied != B then
                fault = TRUE;
            else
                memcpy.fromaddress = memcpy.fromaddress - B;
                memcpy.toaddress = memcpy.toaddress - B;

        if !fault then
            memcpy.cpysize = memcpy.cpysize - B;
            memcpy.stagecpysize = memcpy.stagecpysize - B;

UpdateCpyRegisters(memcpy, fault, copied);

if fault then
    if IsFault(memaddrdesc) then
        AArch64.Abort(memaddrdesc.vaddress, memaddrdesc.fault);
    else
        AccessDescriptor accdesc = if iswrite then waccdesc else raccdesc;
        HandleExternalAbort(memstatus, iswrite, memaddrdesc, B, accdesc);

if memcpy.stage == MOPSStage_Prologue then
    PSTATE.<N,Z,C,V> = memcpy.nzcv;
```

C6.2.104 CPYPWTWN, CPYMWTWN, CPYEWTWN

Memory Copy, writes unprivileged and non-temporal. These instructions perform a memory copy. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYPWTWN, then CPYMWTWN, and then CPYEWTWN.

CPYPWTWN performs some preconditioning of the arguments suitable for using the CPYMWTWN instruction, and performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYMWTWN performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYEWTWN performs the last part of the memory copy.

Note

The inclusion of IMPLEMENTATION DEFINED amounts of memory copy allows some optimization of the size that can be performed.

For CPYPWTWN, the following saturation logic is applied:

If Xn<63:55> != 000000000, the copy size Xn is saturated to `0x007FFFFFFFFFFFFF`.

After that saturation logic is applied, the direction of the memory copy is based on the following algorithm:

If (Xs > Xd) && (Xd + saturated Xn) > Xs, then direction = forward

Elsif (Xs < Xd) && (Xs + saturated Xn) > Xd, then direction = backward

Else direction = IMPLEMENTATION DEFINED choice between forward and backward.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

After execution of CPYPWTWN, option A (which results in encoding PSTATE.C = 0):

- PSTATE.{N,Z,V} are set to {0,0,0}.
- If the copy is in the forward direction, then:
  - Xs holds the original Xs + saturated Xn.
  - Xd holds the original Xd + saturated Xn.
  - Xn holds -1* saturated Xn + an IMPLEMENTATION DEFINED number of bytes copied.
- If the copy is in the backward direction, then:
  - Xs and Xd are unchanged.
  - Xn holds the saturated value of Xn - an IMPLEMENTATION DEFINED number of bytes copied.

After execution of CPYPWTWN, option B (which results in encoding PSTATE.C = 1):

- If the copy is in the forward direction, then:
  - Xs holds the original Xs + an IMPLEMENTATION DEFINED number of bytes copied.
  - Xd holds the original Xd + an IMPLEMENTATION DEFINED number of bytes copied.
  - Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - PSTATE.{N,Z,V} are set to {0,0,0}.
- If the copy is in the backward direction, then:
  - Xs holds the original Xs + saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - Xd holds the original Xd + saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - PSTATE.{N,Z,V} are set to {1,0,0}.

For CPYMWTWN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number.
- If the copy is in the forward direction (Xn is a negative number), then:
  - Xn holds -1* the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the lowest address that the copy is copied from -Xn.
  - Xd holds the lowest address that the copy is made to -Xn.
  - At the end of the instruction, the value of Xn is written back with -1* the number of bytes remaining to be copied in the memory copy in total.
- If the copy is in the backward direction (Xn is a positive number), then:
  - Xn holds the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the highest address that the copy is copied from -Xn+1.
  - Xd holds the highest address that the copy is copied to -Xn+1.
  - At the end of the instruction, the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.

For CPYMWTWN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes to be copied in the memory copy in total.
- If the copy is in the forward direction (PSTATE.N == 0), then:
  - Xs holds the lowest address that the copy is copied from.
  - Xd holds the lowest address that the copy is copied to.
  - At the end of the instruction:
    - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
    - the value of Xs is written back with the lowest address that has not been copied from.
    - the value of Xd is written back with the lowest address that has not been copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
  - Xs holds the highest address that the copy is copied from +1.
  - Xd holds the highest address that the copy is copied to +1.
  - At the end of the instruction:
    - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
    - the value of Xs is written back with the highest address that has not been copied from +1.
    - the value of Xd is written back with the highest address that has not been copied to +1.

For CPYEWTWN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number.
- If the copy is in the forward direction (Xn is a negative number), then:
  - Xn holds -1* the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the lowest address that the copy is copied from -Xn.
  - Xd holds the lowest address that the copy is made to -Xn.
  - At the end of the instruction, the value of Xn is written back with 0.
- If the copy is in the backward direction (Xn is a positive number), then:
  - Xn holds the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the highest address that the copy is copied from -Xn+1.

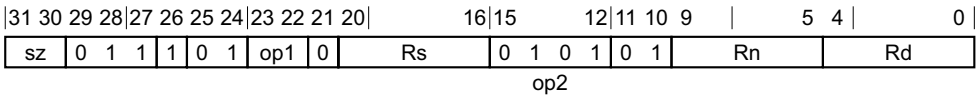
  - Xd holds the highest address that the copy is copied to -Xn+1.
  - At the end of the instruction, the value of Xn is written back with 0.

For CPYEWTWN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes to be copied in the memory copy in total.
- If the copy is in the forward direction (PSTATE.N == 0), then:
  - Xs holds the lowest address that the copy is copied from.
  - Xd holds the lowest address that the copy is copied to.
  - At the end of the instruction:
    - the value of Xn is written back with 0.
    - the value of Xs is written back with the lowest address that has not been copied from.
    - the value of Xd is written back with the lowest address that has not been copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
  - Xs holds the highest address that the copy is copied from +1.
  - Xd holds the highest address that the copy is copied to +1.
  - At the end of the instruction:
    - the value of Xn is written back with 0.
    - the value of Xs is written back with the highest address that has not been copied from +1.
    - the value of Xd is written back with the highest address that has not been copied to +1.

Integer

(FEAT_MOPS)



Epilogue variant

Applies when `op1 == 10`.

```
CPYEWTWN [<Xd>]!, [<Xs>]!, <Xn>!
```

Main variant

Applies when `op1 == 01`.

```
CPYMWTWN [<Xd>]!, [<Xs>]!, <Xn>!
```

Prologue variant

Applies when `op1 == 00`.

```
CPYPWTWN [<Xd>]!, [<Xs>]!, <Xn>!
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_MOPS) || sz != '00' then UNDEFINED;

CPYParams memcpy;
memcpy.d = UInt(Rd);
memcpy.s = UInt(Rs);
memcpy.n = UInt(Rn);
bits(4) options = op2;
```

```
boolean rnontemporal = options<3> == '1';
boolean wnontemporal = options<2> == '1';

case op1 of
    when '00' memcpy.stage = MOPSStage_Prologue;
    when '01' memcpy.stage = MOPSStage_Main;
    when '10' memcpy.stage = MOPSStage_Epilogue;
    otherwise SEE "Memory Copy and Memory Set";

CheckMOPSEnabled();

if (memcpy.s == memcpy.n || memcpy.s == memcpy.d || memcpy.n == memcpy.d || memcpy.d == 31 || memcpy.s
== 31 || memcpy.n == 31) then
    Constraint c = ConstrainUnpredictable(Unpredictable_MOPSOVERLAP31);
    assert c IN {Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNDEF UNDEFINED;
        when Constraint_NOP   EndOfInstruction();
```

Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*, and particularly *Memory Copy and Memory Set CPY**.

Assembler symbols

<Xd> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the "Rd" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.

<Xs> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the "Rs" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the "Rs" field.

<Xn> For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero at the end of the instruction, encoded in the "Rn" field.

For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the "Rn" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the "Rn" field.

Operation

```
constant integer N = MaxBlockSizeCopiedBytes();
bits(8*N) readdata;

memcpy.nzcv = PSTATE.<N,Z,C,V>;
memcpy.toaddress = X[memcpy.d, 64];
memcpy.fromaddress = X[memcpy.s, 64];

if memcpy.stage == MOPSStage_Prologue then
    memcpy.cpysize = UInt(X[memcpy.n, 64]);
else
    memcpy.cpysize = SInt(X[memcpy.n, 64]);

memcpy.implements_option_a = CPYOptionA();
```

```
boolean rprivileged = if options<1> == '1' then AArch64.IsUnprivAccessPriv() else PSTATE.EL != EL0;
boolean wprivileged = if options<0> == '1' then AArch64.IsUnprivAccessPriv() else PSTATE.EL != EL0;

AccessDescriptor raccdesc = CreateAccDescMOPS(MemOp_LOAD, rprivileged, rnontemporal);
AccessDescriptor waccdesc = CreateAccDescMOPS(MemOp_STORE, wprivileged, wnontemporal);

if memcpy.stage == MOPSStage_Prologue then
    if memcpy.cpysize > 0x007FFFFFFFFFFFF then
        memcpy.cpysize = 0x007FFFFFFFFFFFF;

    memcpy.forward = IsMemCpyForward(memcpy);

    if memcpy.implements_option_a then
        memcpy.nzcv = '0000';
        if memcpy.forward then
            // Copy in the forward direction offsets the arguments.
            memcpy.toaddress = memcpy.toaddress + memcpy.cpysize;
            memcpy.fromaddress = memcpy.fromaddress + memcpy.cpysize;
            memcpy.cpysize = 0 - memcpy.cpysize;
    else
        if !memcpy.forward then
            // Copy in the reverse direction offsets the arguments.
            memcpy.toaddress = memcpy.toaddress + memcpy.cpysize;
            memcpy.fromaddress = memcpy.fromaddress + memcpy.cpysize;
            memcpy.nzcv = '1010';
        else
            memcpy.nzcv = '0010';

memcpy.stagecpysize = MemCpyStageSize(memcpy);

if memcpy.stage != MOPSStage_Prologue then
    memcpy.forward = memcpy.cpysize < 0 || (!memcpy.implements_option_a && memcpy.nzcv<3> == '0');
    CheckMemCpyParams(memcpy, options);

integer copied;
boolean iswrite;
AddressDescriptor memaddrdesc;
PhysMemRetStatus memstatus;
boolean fault = FALSE;
integer B;

if memcpy.implements_option_a then
    while memcpy.stagecpysize != 0 && !fault do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(memcpy);

        if memcpy.forward then
            assert B <= -1 * memcpy.stagecpysize;
            (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress + memcpy.cpysize,
memcpy.fromaddress + memcpy.cpysize, memcpy.forward, B, raccdesc, waccdesc);
            if copied != B then
                fault = TRUE;
            else
                memcpy.cpysize = memcpy.cpysize + B;
                memcpy.stagecpysize = memcpy.stagecpysize + B;

        else
            assert B <= memcpy.stagecpysize;
            memcpy.cpysize = memcpy.cpysize - B;
            memcpy.stagecpysize = memcpy.stagecpysize - B;

            (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress + memcpy.cpysize,
memcpy.fromaddress + memcpy.cpysize, memcpy.forward, B, raccdesc, waccdesc);
            if copied != B then
                fault = TRUE;
                memcpy.cpysize = memcpy.cpysize + B;
                memcpy.stagecpysize = memcpy.stagecpysize + B;
```

```
else
    while memcpy.stagecpysize > 0 && !fault do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(memcpy);
        assert B <= memcpy.stagecpysize;

        if memcpy.forward then
            (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress,
memcpy.fromaddress, memcpy.forward, B, raccdesc, waccdesc);
            if copied != B then
                fault = TRUE;
            else
                memcpy.fromaddress = memcpy.fromaddress + B;
                memcpy.toaddress = memcpy.toaddress + B;
        else
            (copied, iswrite, memaddrdesc, memstatus) = MemCpyBytes(memcpy.toaddress - B,
memcpy.fromaddress - B, memcpy.forward, B, raccdesc, waccdesc);

            if copied != B then
                fault = TRUE;
            else
                memcpy.fromaddress = memcpy.fromaddress - B;
                memcpy.toaddress = memcpy.toaddress - B;

        if !fault then
            memcpy.cpysize = memcpy.cpysize - B;
            memcpy.stagecpysize = memcpy.stagecpysize - B;

UpdateCpyRegisters(memcpy, fault, copied);

if fault then
    if IsFault(memaddrdesc) then
        AArch64.Abort(memaddrdesc.vaddress, memaddrdesc.fault);
    else
        AccessDescriptor accdesc = if iswrite then waccdesc else raccdesc;
        HandleExternalAbort(memstatus, iswrite, memaddrdesc, B, accdesc);

if memcpy.stage == MOPSStage_Prologue then
    PSTATE.<N,Z,C,V> = memcpy.nzcv;
```

C6.2.105 CRC32B, CRC32H, CRC32W, CRC32X

CRC32 checksum performs a cyclic redundancy check (CRC) calculation on a value held in a general-purpose register. It takes an input CRC value in the first source operand, performs a CRC on the input value in the second source operand, and returns the output CRC value. The second source operand can be 8, 16, 32, or 64 bits. To align with common usage, the bit order of the values is reversed as part of the operation, and the polynomial 0x04C11DB7 is used for the CRC calculation.

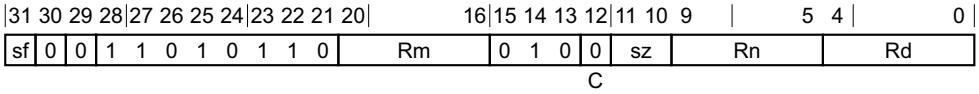
In an Armv8.0 implementation, this is an OPTIONAL instruction. From Armv8.1, it is mandatory for all implementations to implement this instruction.

Note

ID_AA64ISAR0_EL1.CRC32 indicates whether this instruction is supported.

CRC

(FEAT_CRC32)



CRC32B variant

Applies when sf == 0 && sz == 00.

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

CRC32H variant

Applies when sf == 0 && sz == 01.

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

CRC32W variant

Applies when sf == 0 && sz == 10.

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

CRC32X variant

Applies when sf == 1 && sz == 11.

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

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_CRC32) then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sf == '1' && sz != '11' then UNDEFINED;
if sf == '0' && sz == '11' then UNDEFINED;
constant integer size = 8 << UInt(sz);
```

Assembler symbols

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

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

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

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

Operation

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

bits(32+size) tempacc = BitReverse(acc):Zeros(size);
bits(size+32) tempval = BitReverse(val):Zeros(32);

// Poly32Mod2 on a bitstring does a polynomial Modulus over {0,1} operation
X[d, 32] = BitReverse(Poly32Mod2(tempacc EOR tempval, poly));
```

Operational information

If PSTATE.DIT is 1:

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

C6.2.106 CRC32CB, CRC32CH, CRC32CW, CRC32CX

CRC32C checksum performs a cyclic redundancy check (CRC) calculation on a value held in a general-purpose register. It takes an input CRC value in the first source operand, performs a CRC on the input value in the second source operand, and returns the output CRC value. The second source operand can be 8, 16, 32, or 64 bits. To align with common usage, the bit order of the values is reversed as part of the operation, and the polynomial 0x1EDC6F41 is used for the CRC calculation.

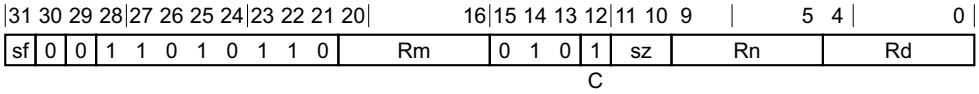
In an Armv8.0 implementation, this is an OPTIONAL instruction. From Armv8.1, it is mandatory for all implementations to implement this instruction.

Note

ID_AA64ISAR0_EL1.CRC32 indicates whether this instruction is supported.

CRC

(FEAT_CRC32)



CRC32CB variant

Applies when sf == 0 && sz == 00.

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

CRC32CH variant

Applies when sf == 0 && sz == 01.

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

CRC32CW variant

Applies when sf == 0 && sz == 10.

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

CRC32CX variant

Applies when sf == 1 && sz == 11.

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

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_CRC32) then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sf == '1' && sz != '11' then UNDEFINED;
if sf == '0' && sz == '11' then UNDEFINED;
constant integer size = 8 << UInt(sz);
```

Assembler symbols

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

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

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

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

Operation

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

bits(32+size) tempacc = BitReverse(acc):Zeros(size);
bits(size+32) tempval = BitReverse(val):Zeros(32);

// Poly32Mod2 on a bitstring does a polynomial Modulus over {0,1} operation
X[d, 32] = BitReverse(Poly32Mod2(tempacc EOR tempval, poly));
```

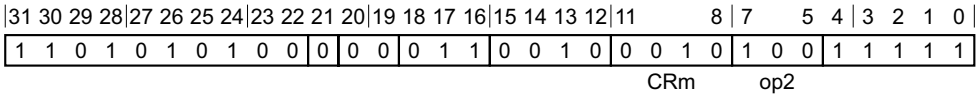
Operational information

If PSTATE.DIT is 1:

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

C6.2.107 CSDB

Consumption of Speculative Data Barrier is a memory barrier that controls speculative execution arising from data value prediction. For more information and details of the semantics, see *Consumption of Speculative Data Barrier (CSDB)*.



Encoding

CSDB

Decode for this encoding

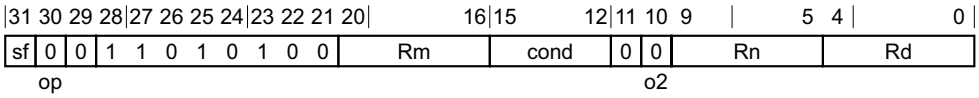
```
// Empty.
```

Operation

```
ConsumptionOfSpeculativeDataBarrier();
```

C6.2.108 CSEL

If the condition is true, Conditional Select writes the value of the first source register to the destination register. If the condition is false, it writes the value of the second source register to the destination register.



32-bit variant

Applies when `sf == 0`.

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

64-bit variant

Applies when `sf == 1`.

```
CSEL <Xd>, <Xn>, <Xm>, <cond>
```

Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
constant integer datasize = 32 << UInt(sf);
```

Assembler symbols

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

Operation

```
bits(datasize) result;
if ConditionHolds(cond) then
    result = X[n, datasize];
else
    result = X[m, datasize];

X[d, datasize] = result;
```

Operational information

If PSTATE.DIT is 1:

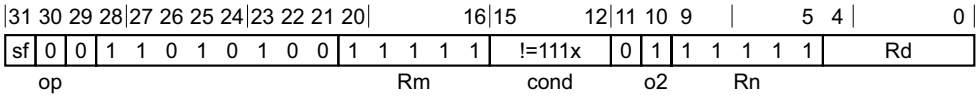
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.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

C6.2.109 CSET

Conditional Set sets the destination register to 1 if the condition is TRUE, and otherwise sets it to 0.

This instruction is an alias of the CSINC instruction. This means that:

- The encodings in this description are named to match the encodings of CSINC.
- The description of CSINC gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.



32-bit variant

Applies when `sf == 0`.

```
CSET <Wd>, <cond>
```

is equivalent to

```
CSINC <Wd>, WZR, WZR, invert(<cond>)
```

and is always the preferred disassembly.

64-bit variant

Applies when `sf == 1`.

```
CSET <Xd>, <cond>
```

is equivalent to

```
CSINC <Xd>, XZR, XZR, invert(<cond>)
```

and is always the preferred disassembly.

Assembler symbols

`<Wd>`	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
`<Xd>`	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
`<cond>`	Is one of the standard conditions, excluding AL and NV, encoded in the "cond" field with its least significant bit inverted.

Operation

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

Operational information

If PSTATE.DIT is 1:

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

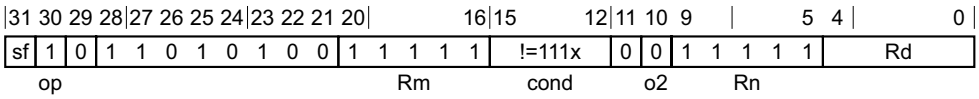
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

C6.2.110 CSETM

Conditional Set Mask sets all bits of the destination register to 1 if the condition is TRUE, and otherwise sets all bits to 0.

This instruction is an alias of the CSINV instruction. This means that:

- The encodings in this description are named to match the encodings of CSINV.
- The description of CSINV gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.



32-bit variant

Applies when `sf == 0`.

```
CSETM <Wd>, <cond>
```

is equivalent to

```
CSINV <Wd>, WZR, WZR, invert(<cond>)
```

and is always the preferred disassembly.

64-bit variant

Applies when `sf == 1`.

```
CSETM <Xd>, <cond>
```

is equivalent to

```
CSINV <Xd>, XZR, XZR, invert(<cond>)
```

and is always the preferred disassembly.

Assembler symbols

`<Wd>`	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
`<Xd>`	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
`<cond>`	Is one of the standard conditions, excluding AL and NV, encoded in the "cond" field with its least significant bit inverted.

Operation

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

Operational information

If PSTATE.DIT is 1:

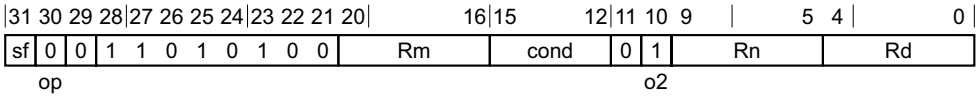
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

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

C6.2.111 CSINC

Conditional Select Increment returns, in the destination register, the value of the first source register if the condition is TRUE, and otherwise returns the value of the second source register incremented by 1.

This instruction is used by the aliases CINC and CSET. See *Alias conditions* for details of when each alias is preferred.



32-bit variant

Applies when `sf == 0`.

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

64-bit variant

Applies when `sf == 1`.

```
CSINC <Xd>, <Xn>, <Xm>, <cond>
```

Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
constant integer datasize = 32 << UInt(sf);
```

Alias conditions

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

Assembler symbols

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

Operation

```
bits(datasize) result;
if ConditionHolds(cond) then
    result = X[n, datasize];
else
    result = X[m, datasize];
    result = result + 1;

X[d, datasize] = result;
```

Operational information

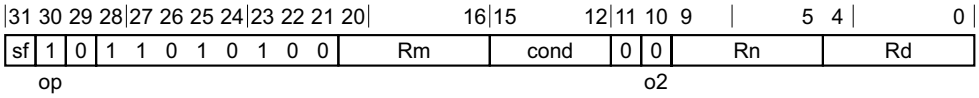
If PSTATE.DIT is 1:

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

C6.2.112 CSINV

Conditional Select Invert returns, in the destination register, the value of the first source register if the condition is TRUE, and otherwise returns the bitwise inversion value of the second source register.

This instruction is used by the aliases CINV and CSETM. See *Alias conditions* for details of when each alias is preferred.



32-bit variant

Applies when `sf == 0`.

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

64-bit variant

Applies when `sf == 1`.

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

Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
constant integer datasize = 32 << UInt(sf);
```

Alias conditions

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

Assembler symbols

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

Operation

```
bits(datasize) result;
if ConditionHolds(cond) then
    result = X[n, datasize];
else
    result = X[m, datasize];
    result = NOT(result);

X[d, datasize] = result;
```

Operational information

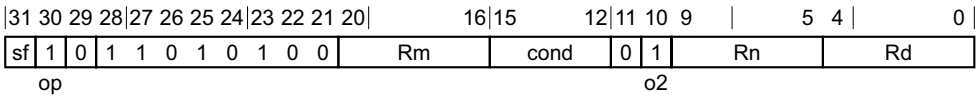
If PSTATE.DIT is 1:

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

C6.2.113 CSNEG

Conditional Select Negation returns, in the destination register, the value of the first source register if the condition is TRUE, and otherwise returns the negated value of the second source register.

This instruction is used by the alias CNEG. See *Alias conditions* for details of when each alias is preferred.



32-bit variant

Applies when `sf == 0`.

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

64-bit variant

Applies when `sf == 1`.

```
CSNEG <Xd>, <Xn>, <Xm>, <cond>
```

Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
constant integer datasize = 32 << UInt(sf);
```

Alias conditions

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

Assembler symbols

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

Operation

```
bits(datasize) result;
if ConditionHolds(cond) then
    result = X[n, datasize];
else
```

```
    result = X[m, datasize];
    result = NOT(result);
    result = result + 1;

X[d, datasize] = result;
```

Operational information

If PSTATE.DIT is 1:

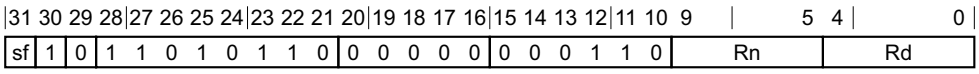
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.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

C6.2.114 CTZ

Count Trailing Zeros counts the number of consecutive binary zero bits, starting from the least significant bit in the source register, and places the count in the destination register.

Integer

(FEAT_CSSC)



32-bit variant

Applies when `sf == 0`.

```
CTZ <Wd>, <Wn>
```

64-bit variant

Applies when `sf == 1`.

```
CTZ <Xd>, <Xn>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_CSSC) then UNDEFINED;
constant integer datasize = 32 << UInt(sf);
integer n = UInt(Rn);
integer d = UInt(Rd);
```

Assembler symbols

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

Operation

```
bits(datasize) operand1 = X[n, datasize];
integer result = CountLeadingZeroBits(BitReverse(operand1));
X[d, datasize] = result<datasize-1:0>;
```

Operational information

If PSTATE.DIT is 1:

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

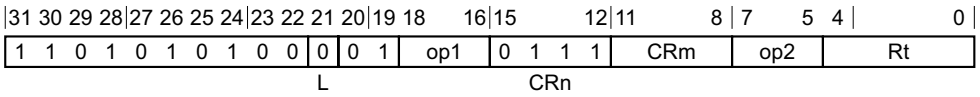
— The values of the NZCV flags.

C6.2.115 DC

Data Cache operation. For more information, see *op0==0b01, cache maintenance, TLB maintenance, address translation, prediction restriction, BRBE, Trace Extension, and Guarded Control Stack instructions*.

This instruction is an alias of the SYS instruction. This means that:

- The encodings in this description are named to match the encodings of SYS.
- The description of SYS gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.



Encoding

```
DC <dc_op>, <Xt>
```

is equivalent to

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

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

Assembler symbols

`<dc_op>` Is a DC instruction name, as listed for the DC system instruction group, encoded in the "op1:CRm:op2" field. It can have the following values:

`IVAC`	when `op1 = 000, CRm = 0110, op2 = 001`
`ISW`	when `op1 = 000, CRm = 0110, op2 = 010`
`CSW`	when `op1 = 000, CRm = 1010, op2 = 010`
`CISW`	when `op1 = 000, CRm = 1110, op2 = 010`
`ZVA`	when `op1 = 011, CRm = 0100, op2 = 001`
`CVAC`	when `op1 = 011, CRm = 1010, op2 = 001`
`CVAU`	when `op1 = 011, CRm = 1011, op2 = 001`
`CIVAC`	when `op1 = 011, CRm = 1110, op2 = 001`

When FEAT_MTE2 is implemented, the following values are also valid:

`IGVAC`	when `op1 = 000, CRm = 0110, op2 = 011`
`IGSW`	when `op1 = 000, CRm = 0110, op2 = 100`
`IGDVAC`	when `op1 = 000, CRm = 0110, op2 = 101`
`IGDSW`	when `op1 = 000, CRm = 0110, op2 = 110`
`CGSW`	when `op1 = 000, CRm = 1010, op2 = 100`
`CGDSW`	when `op1 = 000, CRm = 1010, op2 = 110`
`CIGSW`	when `op1 = 000, CRm = 1110, op2 = 100`
`CIGDSW`	when `op1 = 000, CRm = 1110, op2 = 110`

When FEAT_MTE is implemented, the following values are also valid:

`GVA`	when `op1 = 011, CRm = 0100, op2 = 011`
`GZVA`	when `op1 = 011, CRm = 0100, op2 = 100`

CGVAC	when op1 = 011, CRm = 1010, op2 = 011
CGDVAC	when op1 = 011, CRm = 1010, op2 = 101
CGVAP	when op1 = 011, CRm = 1100, op2 = 011
CGDVAP	when op1 = 011, CRm = 1100, op2 = 101
CGVADP	when op1 = 011, CRm = 1101, op2 = 011
CGDVADP	when op1 = 011, CRm = 1101, op2 = 101
CIGVAC	when op1 = 011, CRm = 1110, op2 = 011
CIGDVAC	when op1 = 011, CRm = 1110, op2 = 101

When FEAT_DPB is implemented, the following value is also valid:

CVAP	when op1 = 011, CRm = 1100, op2 = 001

When FEAT_DPB2 is implemented, the following value is also valid:

CVADP	when op1 = 011, CRm = 1101, op2 = 001

When FEAT_MEC is implemented, the following values are also valid:

CIPAE	when op1 = 100, CRm = 1110, op2 = 000
CIGDPAE	when op1 = 100, CRm = 1110, op2 = 111

When FEAT_RME is implemented, the following values are also valid:

CIPAPA	when op1 = 110, CRm = 1110, op2 = 001
CIGDPAPA	when op1 = 110, CRm = 1110, op2 = 101

<op1>	Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op1" field.
<Cm>	Is a name 'Cm', with 'm' in the range 0 to 15, encoded in the "CRm" field.
<op2>	Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op2" field.
<Xt>	Is the 64-bit name of the general-purpose source register, encoded in the "Rt" field.

Operation

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

C6.2.116 DCPS1

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

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

The target exception level of a DCPS1 instruction is:

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

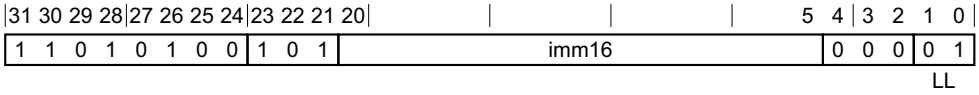
When the target Exception level of a DCPS1 instruction is ELx, on executing this instruction:

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

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

This instruction is always UNDEFINED in Non-debug state.

For more information on the operation of the DCPS<n> instructions, see *DCPS<n>*.



Encoding

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

Decode for this encoding

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

Assembler symbols

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

Operation

```
DCPSInstruction(LL);
```

C6.2.117 DCPS2

Debug Change PE State to EL2, when executed in Debug state:

- If executed at EL0 or EL1 changes the current Exception level and SP to EL2 using SP_EL2.
- Otherwise, if executed at ELx, selects SP_ELx.

The target exception level of a DCPS2 instruction is:

- EL2 if the instruction is executed at an exception level that is not EL3.
- EL3 if the instruction is executed at EL3.

When the target Exception level of a DCPS2 instruction is ELx, on executing this instruction:

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

This instruction is UNDEFINED at the following exception levels:

- All exception levels if EL2 is not implemented.
- At EL0 and EL1 if EL2 is disabled in the current Security state.

This instruction is always UNDEFINED in Non-debug state.

For more information on the operation of the DCPS<n> instructions, see *DCPS<n>*.

Encoding

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

Decode for this encoding

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

Assembler symbols

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

Operation

```
DCPSInstruction(LL);
```

C6.2.118 DCPS3

Debug Change PE State to EL3, when executed in Debug state:

- If executed at EL3 selects SP_EL3.
- Otherwise, changes the current Exception level and SP to EL3 using SP_EL3.

The target exception level of a DCPS3 instruction is EL3.

On executing a DCPS3 instruction:

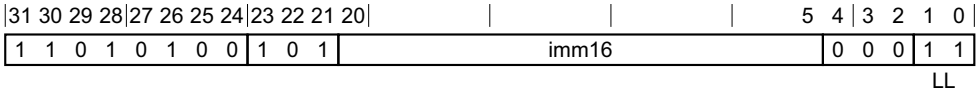
- ELR_EL3 becomes UNKNOWN.
- SPSR_EL3 becomes UNKNOWN.
- ESR_EL3 becomes UNKNOWN.
- DLR_EL0 and DSPSR_EL0 become UNKNOWN.
- The endianness is set according to SCTLR_EL3.EE.

This instruction is UNDEFINED at all exception levels if either:

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

This instruction is always UNDEFINED in Non-debug state.

For more information on the operation of the DCPS<n> instructions, see *DCPS<n>*.



Encoding

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

Decode for this encoding

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

Assembler symbols

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

Operation

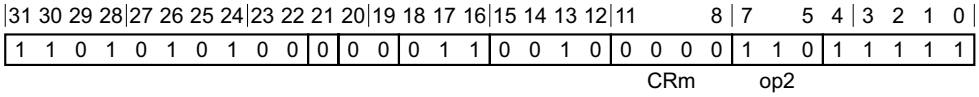
```
DCPSInstruction(LL);
```

C6.2.119 DGH

Data Gathering Hint is a hint instruction that indicates that it is not expected to be performance optimal to merge memory accesses with Normal Non-cacheable or Device-GRE attributes appearing in program order before the hint instruction with any memory accesses appearing after the hint instruction into a single memory transaction on an interconnect.

System

(FEAT_DGH)



Encoding

DGH

Decode for this encoding

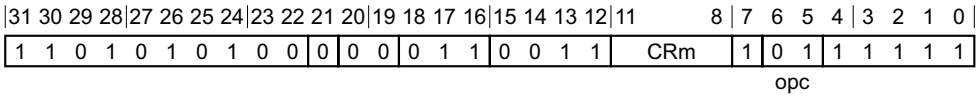
```
if !IsFeatureImplemented(FEAT_DGH) then EndOfInstruction();
```

Operation

```
Hint_DGH();
```

C6.2.120 DMB

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



Encoding

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

Decode for this encoding

```
MBReqDomain domain;
MBReqTypes types;
case CRm<3:2> of
    when '00' domain = MBReqDomain_OuterShareable;
    when '01' domain = MBReqDomain_Nonshareable;
    when '10' domain = MBReqDomain_InnerShareable;
    when '11' domain = MBReqDomain_FullSystem;
case CRm<1:0> of
    when '00' types = MBReqTypes_All; domain = MBReqDomain_FullSystem;
    when '01' types = MBReqTypes_Reads;
    when '10' types = MBReqTypes_Writes;
    when '11' types = MBReqTypes_All;
```

Assembler symbols

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

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

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

`LD` Full system is the required shareability domain, reads are the required access type before the barrier instruction, and reads and writes are the required access types after the barrier instruction. Encoded as CRm = `0b1101`.

`ISH` Inner Shareable is the required shareability domain, reads and writes are the required access types, both before and after the barrier instruction. Encoded as CRm = `0b1011`.

`ISHST` Inner Shareable is the required shareability domain, writes are the required access type, both before and after the barrier instruction. Encoded as CRm = `0b1010`.

`ISHLD` Inner Shareable is the required shareability domain, reads are the required access type before the barrier instruction, and reads and writes are the required access types after the barrier instruction. Encoded as CRm = `0b1001`.

`NSH` Non-shareable is the required shareability domain, reads and writes are the required access, both before and after the barrier instruction. Encoded as CRm = `0b0111`.

`NSHST` Non-shareable is the required shareability domain, writes are the required access type, both before and after the barrier instruction. Encoded as CRm = `0b0110`.

`NSHLD` Non-shareable is the required shareability domain, reads are the required access type before the barrier instruction, and reads and writes are the required access types after the barrier instruction. Encoded as CRm = `0b0101`.

OSH — Outer Shareable is the required shareability domain, reads and writes are the required access types, both before and after the barrier instruction. Encoded as CRm = 0b0011.

OSHST — Outer Shareable is the required shareability domain, writes are the required access type, both before and after the barrier instruction. Encoded as CRm = 0b0010.

OSHLD — Outer Shareable is the required shareability domain, reads are the required access type before the barrier instruction, and reads and writes are the required access types after the barrier instruction. Encoded as CRm = 0b0001.

All other encodings of CRm that are not listed above are reserved, and can be encoded using the #<imm> syntax. All unsupported and reserved options must execute as a full system barrier operation, but software must not rely on this behavior. For more information on whether an access is before or after a barrier instruction, see *Data Memory Barrier (DMB)* or see *Data Synchronization Barrier (DSB)*.

<imm> Is a 4-bit unsigned immediate, in the range 0 to 15, encoded in the "CRm" field.

Operation

```
DataMemoryBarrier(domain, types);
```

C6.2.121 DRPS

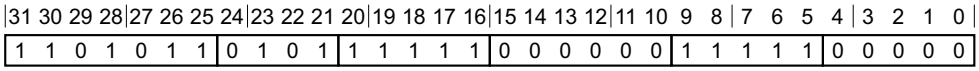
Debug restore PE state using the SPSR for the current Exception level. When executed, the PE restores PSTATE from the SPSR.

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

This instruction is UNDEFINED at EL0.

This instruction is UNDEFINED in Non-debug state.

For more information on the operation of DRPS, see *DRPS*.



Encoding

DRPS

Decode for this encoding

```
if !Halted() || PSTATE.EL == EL0 then UNDEFINED;
```

Operation

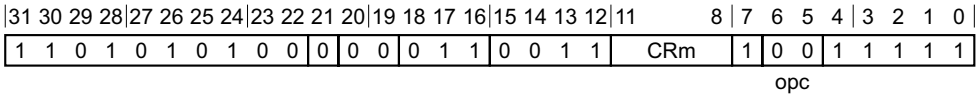
```
DRPSInstruction();
```

C6.2.122 DSB

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

This instruction is used by the aliases PSSBB and SSBB. See *Alias conditions* for details of when each alias is preferred.

Memory barrier



Encoding

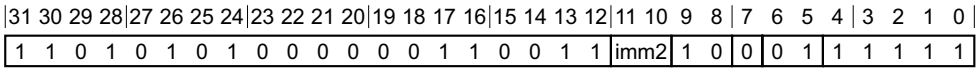
```
DSB <option>|#<imm>
```

Decode for this encoding

```
boolean nXS = FALSE;

DSBAlias alias;
case CRm of
    when '0000' alias = DSBAlias_SSBB;
    when '0100' alias = DSBAlias_PSSBB;
    otherwise alias = DSBAlias_DSB;

MBReqDomain domain;
case CRm<3:2> of
    when '00' domain = MBReqDomain_OuterShareable;
    when '01' domain = MBReqDomain_Nonshareable;
    when '10' domain = MBReqDomain_InnerShareable;
    when '11' domain = MBReqDomain_FullSystem;

MBReqTypes types;
case CRm<1:0> of
    when '00' types = MBReqTypes_All; domain = MBReqDomain_FullSystem;
    when '01' types = MBReqTypes_Reads;
    when '10' types = MBReqTypes_Writes;
    when '11' types = MBReqTypes_All;
```

Memory nXS barrier

(FEAT_XS)



Encoding

```
DSB <option>nXS
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_XS) then UNDEFINED;
MBReqTypes types = MBReqTypes_All;
boolean nXS = TRUE;
DSBAlias alias = DSBAlias_DSB;
MBReqDomain domain;
```

```
case imm2 of
    when '00' domain = MBReqDomain_OuterShareable;
    when '01' domain = MBReqDomain_Nonshareable;
    when '10' domain = MBReqDomain_InnerShareable;
    when '11' domain = MBReqDomain_FullSystem;
```

Alias conditions

Alias	is preferred when
PSSBB	CRm == '0100'
SSBB	CRm == '0000'

Assembler symbols

<option> For the memory barrier variant: specifies the limitation on the barrier operation. Values are:

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

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

LD Full system is the required shareability domain, reads are the required access type before the barrier instruction, and reads and writes are the required access types after the barrier instruction. Encoded as CRm = 0b1101.

ISH Inner Shareable is the required shareability domain, reads and writes are the required access types, both before and after the barrier instruction. Encoded as CRm = 0b1011.

ISHST Inner Shareable is the required shareability domain, writes are the required access type, both before and after the barrier instruction. Encoded as CRm = 0b1010.

ISHLD Inner Shareable is the required shareability domain, reads are the required access type before the barrier instruction, and reads and writes are the required access types after the barrier instruction. Encoded as CRm = 0b1001.

NSH Non-shareable is the required shareability domain, reads and writes are the required access, both before and after the barrier instruction. Encoded as CRm = 0b0111.

NSHST Non-shareable is the required shareability domain, writes are the required access type, both before and after the barrier instruction. Encoded as CRm = 0b0110.

NSHLD Non-shareable is the required shareability domain, reads are the required access type before the barrier instruction, and reads and writes are the required access types after the barrier instruction. Encoded as CRm = 0b0101.

OSH Outer Shareable is the required shareability domain, reads and writes are the required access types, both before and after the barrier instruction. Encoded as CRm = 0b0011.

OSHST Outer Shareable is the required shareability domain, writes are the required access type, both before and after the barrier instruction. Encoded as CRm = 0b0010.

OSHLD Outer Shareable is the required shareability domain, reads are the required access type before the barrier instruction, and reads and writes are the required access types after the barrier instruction. Encoded as CRm = 0b0001.

All other encodings of "CRm", other than the values 0b0000 and 0b0100, that are not listed above are reserved, and can be encoded using the #<imm> syntax. All unsupported and reserved options must execute as a full system barrier operation, but software must not rely on this behavior. For more information on whether an access is before or after a barrier instruction, see *Data Memory Barrier (DMB)* or see *Data Synchronization Barrier (DSB)*.

Note

The value 0b0000 is used to encode SSBB and the value 0b0100 is used to encode PSSBB.

For the memory nXS barrier variant: specifies the limitation on the barrier operation. Values are:

SY — Full system is the required shareability domain, reads and writes are the required access types, both before and after the barrier instruction. This option is referred to as the full system barrier. Encoded as imm2 = 0b11.

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 imm2 = 0b10.

NSH — Non-shareable is the required shareability domain, reads and writes are the required access, both before and after the barrier instruction. Encoded as imm2 = 0b01.

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 imm2 = 0b00.

<imm> Is a 4-bit unsigned immediate, in the range 0 to 15, encoded in the "CRm" field.

Operation for all encodings

```
case alias of
    when DSBAlias_SSBB
        SpeculativeStoreBypassBarrierToVA();
    when DSBAlias_PSSBB
        SpeculativeStoreBypassBarrierToPA();
    when DSBAlias_DSB
        if IsFeatureImplemented(FEAT_TME) && TSTATE.depth > 0 then
            FailTransaction(TMFailure_ERR, FALSE);
        if !nXS && IsFeatureImplemented(FEAT_XS) then
            nXS = PSTATE.EL IN {EL0, EL1} && IsHCRXEL2Enabled() && HCRX_EL2.FnXS == '1';
        DataSynchronizationBarrier(domain, types, nXS);
    otherwise
        Unreachable();
```

C6.2.123 DVP

Data Value Prediction Restriction by Context prevents data value predictions that predict execution addresses based on information gathered from earlier execution within a particular execution context. Data value predictions determined by the actions of code in the target execution context or contexts appearing in program order before the instruction cannot be used to exploitatively control speculative execution occurring after the instruction is complete and synchronized.

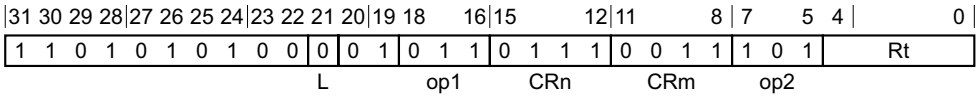
For more information, see DVP RCTX.

This instruction is an alias of the SYS instruction. This means that:

- The encodings in this description are named to match the encodings of SYS.
- The description of SYS gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

System

(FEAT_SPECRES)



Encoding

```
DVP RCTX, <Xt>
```

is equivalent to

```
SYS #3, C7, C3, #5, <Xt>
```

and is always the preferred disassembly.

Assembler symbols

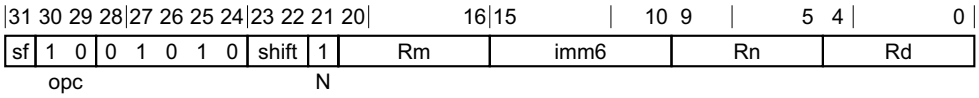
<Xt> Is the 64-bit name of the general-purpose source register, encoded in the "Rt" field.

Operation

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

C6.2.124 EON (shifted register)

Bitwise Exclusive-OR NOT (shifted register) performs a bitwise exclusive-OR NOT of a register value and an optionally-shifted register value, and writes the result to the destination register.



32-bit variant

Applies when `sf == 0`.

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

64-bit variant

Applies when `sf == 1`.

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

Decode for all variants of this encoding

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

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

Assembler symbols

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

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

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

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

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

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

`<shift>` Is the optional shift to be applied to the final source, defaulting to LSL and encoded in the "shift" field. It can have the following values:

- `LSL` when `shift = 00`
- `LSR` when `shift = 01`
- `ASR` when `shift = 10`
- `ROR` when `shift = 11`

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

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

Operation

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

operand2 = NOT(operand2);

result = operand1 EOR operand2;

X[d, datasize] = result;
```

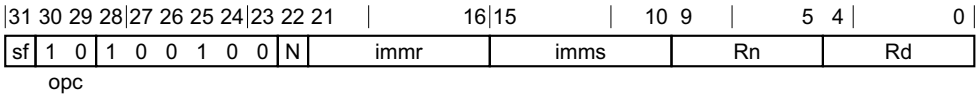
Operational information

If PSTATE.DIT is 1:

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

C6.2.125 EOR (immediate)

Bitwise Exclusive-OR (immediate) performs a bitwise exclusive-OR of a register value and an immediate value, and writes the result to the destination register.



32-bit variant

Applies when `sf == 0 && N == 0`.

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

64-bit variant

Applies when `sf == 1`.

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

Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
constant integer datasize = 32 << UInt(sf);
bits(datasize) imm;
if sf == '0' && N != '0' then UNDEFINED;
(imm, -) = DecodeBitMasks(N, imms, immr, TRUE, datasize);
```

Assembler symbols

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

Operation

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

result = operand1 EOR imm;

if d == 31 then
    SP[] = ZeroExtend(result, 64);
else
    X[d, datasize] = result;
```

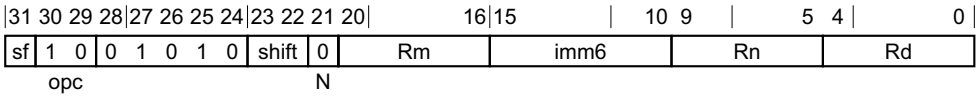
Operational information

If PSTATE.DIT is 1:

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

C6.2.126 EOR (shifted register)

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



32-bit variant

Applies when `sf == 0`.

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

64-bit variant

Applies when `sf == 1`.

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

Decode for all variants of this encoding

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

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

Assembler symbols

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

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

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

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

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

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

`<shift>` Is the optional shift to be applied to the final source, defaulting to LSL and encoded in the "shift" field. It can have the following values:

- `LSL` when `shift = 00`
- `LSR` when `shift = 01`
- `ASR` when `shift = 10`
- `ROR` when `shift = 11`

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

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

Operation

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

result = operand1 EOR operand2;

X[d, datasize] = result;
```

Operational information

If PSTATE.DIT is 1:

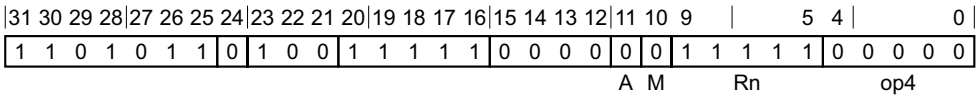
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.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

C6.2.127 ERET

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

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

ERET is UNDEFINED at EL0.



Encoding

ERET

Decode for this encoding

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

Operation

```
AArch64.CheckForERetTrap(FALSE, TRUE);
bits(64) target = ELR_ELx[];

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

C6.2.128 ERETAA, ERETAB

Exception Return, with pointer authentication. This instruction authenticates the address in ELR, using SP as the modifier and the specified key, the PE restores PSTATE from the SPSR for the current Exception level, and branches to the authenticated address.

Key A is used for ERETAA. Key B is used for ERETAB.

If the authentication passes, the PE continues execution at the target of the branch. For information on behavior if the authentication fails, see *Faulting on pointer authentication*.

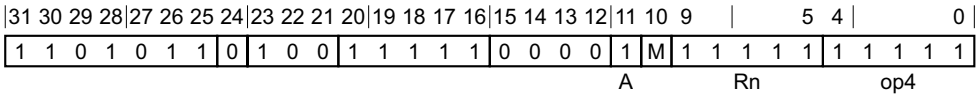
The authenticated address is not written back to ELR.

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

ERETAA and ERETAB are UNDEFINED at EL0.

Integer

(FEAT_PAuth)



ERETAA variant

Applies when M == 0.

ERETAA

ERETAB variant

Applies when M == 1.

ERETAB

Decode for all variants of this encoding

```
if PSTATE.EL == EL0 then UNDEFINED;
boolean use_key_a = (M == '0');

if !IsFeatureImplemented(FEAT_PAuth) then
    UNDEFINED;
```

Operation

```
AArch64.CheckForERetTrap(TRUE, use_key_a);
bits(64) target = ELR_ELx[];
bits(64) modifier = SP[];

if use_key_a then
    target = AuthIA(target, modifier, TRUE);
else
    target = AuthIB(target, modifier, TRUE);

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

C6.2.129 ESB

Error Synchronization Barrier is an error synchronization event that might also update DISR_EL1 and VDISR_EL2.

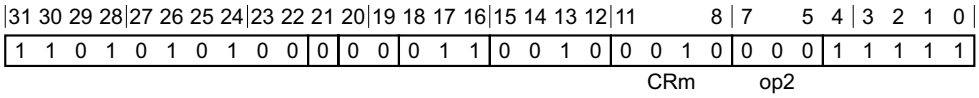
This instruction can be used at all Exception levels and in Debug state.

In Debug state, this instruction behaves as if SError interrupts are masked at all Exception levels. See Error Synchronization Barrier in the Arm(R) Reliability, Availability, and Serviceability (RAS) Specification, Armv8, for Armv8-A architecture profile.

If the RAS Extension is not implemented, this instruction executes as a `NOP`.

System

(FEAT_RAS)



Encoding

`ESB`

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_RAS) then EndOfInstruction();
```

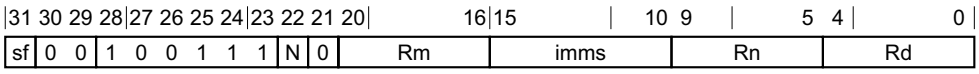
Operation

```
if IsFeatureImplemented(FEAT_TME) && TSTATE.depth > 0 then
    FailTransaction(TMFailure_ERR, FALSE);
SynchronizeErrors();
AArch64.ESBOperation();
if PSTATE.EL IN {EL0, EL1} && EL2Enabled() then AArch64.vESBOperation();
TakeUnmaskedSErrorInterrupts();
```

C6.2.130 EXTR

Extract register extracts a register from a pair of registers.

This instruction is used by the alias ROR (immediate). See *Alias conditions* for details of when each alias is preferred.



32-bit variant

Applies when `sf == 0 && N == 0 && imms == 0xxxxx`.

```
EXTR <Wd>, <Wn>, <Wm>, #<lsb>
```

64-bit variant

Applies when `sf == 1 && N == 1`.

```
EXTR <Xd>, <Xn>, <Xm>, #<lsb>
```

Decode for all variants of this encoding

```
if N != sf then UNDEFINED;
if sf == '0' && imms<5> == '1' then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
constant integer datasize = 32 << UInt(sf);
constant integer lsb = UInt(imms);
```

Alias conditions

Alias	is preferred when
ROR (immediate)	`Rn == Rm`

Assembler symbols

`<Wd>`	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
`<Wn>`	Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
`<Wm>`	Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
`<Xd>`	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
`<Xn>`	Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
`<Xm>`	Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.
`<lsb>`	For the 32-bit variant: is the least significant bit position from which to extract, in the range 0 to 31, encoded in the "imms" field. For the 64-bit variant: is the least significant bit position from which to extract, in the range 0 to 63, encoded in the "imms" field.

Operation

```
bits(datasize) result;
bits(datasize) operand1 = X[n, datasize];
bits(datasize) operand2 = X[m, datasize];
bits(2*datasize) concat = operand1:operand2;

result = concat<(lsb+datasize)-1:lsb>;

X[d, datasize] = result;
```

Operational information

If PSTATE.DIT is 1:

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

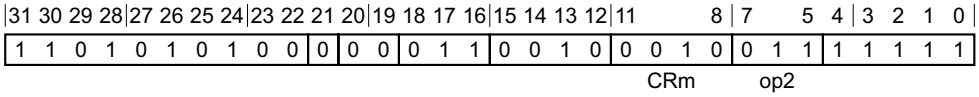
C6.2.131 GCSB

Guarded Control Stack Barrier. This instruction generates a GCSB effect.

If FEAT_GCS is not implemented, this instruction executes as a NOP.

System

(FEAT_GCS)



Encoding

GCSB DSYNC

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_GCS) then EndOfInstruction();
```

Operation

```
GCSSynchronizationBarrier();
```

C6.2.132 GCSPOPCX

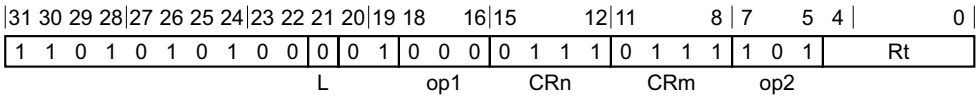
Guarded Control Stack Pop and Compare exception return record loads an exception return record from the location indicated by the current Guarded control stack pointer register, compares the loaded values with the current ELR_ELx, SPSR_ELx, and LR, and increments the pointer by the size of a Guarded control stack exception return record.

This instruction is an alias of the SYS instruction. This means that:

- The encodings in this description are named to match the encodings of SYS.
- The description of SYS gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

System

(FEAT_GCS)



Encoding

```
GCSPOPCX {<Xt>}
```

is equivalent to

```
SYS #0, C7, C7, #5{, <Xt>}
```

and is always the preferred disassembly.

Assembler symbols

<Xt> Is the 64-bit name of the optional general-purpose source register, defaulting to '11111', encoded in the "Rt" field.

Operation

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

C6.2.133 GCSPOPM

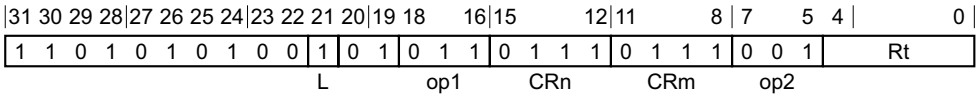
Guarded Control Stack Pop loads the 64-bit doubleword that is pointed to by the current Guarded control stack pointer, writes it to the destination register, and increments the current Guarded control stack pointer register by the size of a Guarded control stack procedure return record.

This instruction is an alias of the SYSL instruction. This means that:

- The encodings in this description are named to match the encodings of SYSL.
- The description of SYSL gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

System

(FEAT_GCS)



Encoding

`GCSPOPM <Xt>`

is equivalent to

`SYSL <Xt>, #3, C7, C7, #1`

and is always the preferred disassembly.

Assembler symbols

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

Operation

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

C6.2.134 GCSPOPX

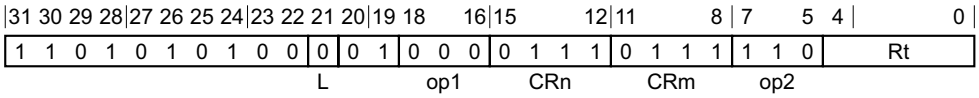
Guarded Control Stack Pop exception return record loads an exception return record from the location indicated by the current Guarded control stack pointer register, checks that the record is an exception return record, and increments the pointer by the size of a Guarded control stack exception return record.

This instruction is an alias of the SYS instruction. This means that:

- The encodings in this description are named to match the encodings of SYS.
- The description of SYS gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

System

(FEAT_GCS)



Encoding

```
GCSPOPX {<Xt>}
```

is equivalent to

```
SYS #0, C7, C7, #6{, <Xt>}
```

and is always the preferred disassembly.

Assembler symbols

<Xt> Is the 64-bit name of the optional general-purpose source register, defaulting to '11111', encoded in the "Rt" field.

Operation

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

C6.2.135 GCSPUSHM

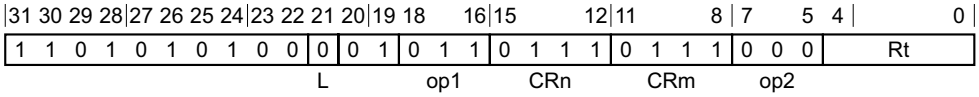
Guarded Control Stack Push decrements the current Guarded control stack pointer register by the size of a Guarded control procedure return record and stores an entry to the Guarded control stack.

This instruction is an alias of the SYS instruction. This means that:

- The encodings in this description are named to match the encodings of SYS.
- The description of SYS gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

System

(FEAT_GCS)



Encoding

`GCSPUSHM <Xt>`

is equivalent to

`SYS #3, C7, C7, #0, <Xt>`

and is always the preferred disassembly.

Assembler symbols

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

Operation

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

C6.2.136 GCSPUSHX

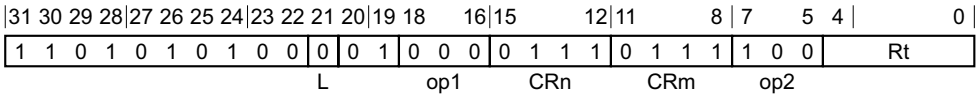
Guarded Control Stack Push exception return record decrements the current Guarded control stack pointer register by the size of a Guarded control stack exception return record and stores an exception return record to the Guarded control stack.

This instruction is an alias of the SYS instruction. This means that:

- The encodings in this description are named to match the encodings of SYS.
- The description of SYS gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

System

(FEAT_GCS)



Encoding

```
GCSPUSHX {<Xt>}
```

is equivalent to

```
SYS #0, C7, C7, #4{, <Xt>}
```

and is always the preferred disassembly.

Assembler symbols

<Xt> Is the 64-bit name of the optional general-purpose source register, defaulting to '11111', encoded in the "Rt" field.

Operation

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

C6.2.137 GCSSS1

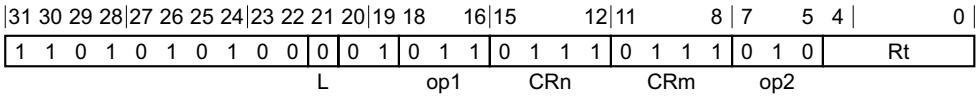
Guarded Control Stack Switch Stack 1 validates that the stack being switched to contains a Valid cap entry, stores an In-progress cap entry to the stack that is being switched to, and sets the current Guarded control stack pointer to the stack that is being switched to.

This instruction is an alias of the SYS instruction. This means that:

- The encodings in this description are named to match the encodings of SYS.
- The description of SYS gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

System

(FEAT_GCS)



Encoding

`GCSSS1 <Xt>`

is equivalent to

`SYS #3, C7, C7, #2, <Xt>`

and is always the preferred disassembly.

Assembler symbols

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

Operation

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

C6.2.138 GCSSS2

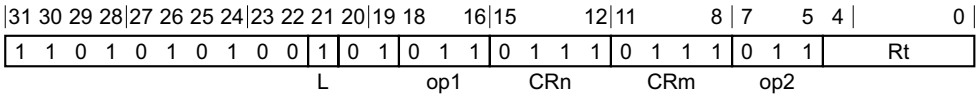
Guarded Control Stack Switch Stack 2 validates that the most recent entry of the Guarded control stack being switched to contains an In-progress cap entry, stores a Valid cap entry to the Guarded control stack that is being switched from, and sets Xt to the Guarded control stack pointer that is being switched from.

This instruction is an alias of the SYSL instruction. This means that:

- The encodings in this description are named to match the encodings of SYSL.
- The description of SYSL gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

System

(FEAT_GCS)



Encoding

`GCSSS2 <Xt>`

is equivalent to

`SYSL <Xt>, #3, C7, C7, #3`

and is always the preferred disassembly.

Assembler symbols

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

Operation

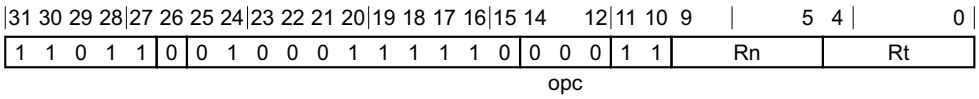
The description of SYSL gives the operational pseudocode for this instruction.

C6.2.139 GCSSTR

Guarded Control Stack Store stores a doubleword from a register to memory. The address that is used for the store is calculated from a base register.

Integer

(FEAT_GCS)



Encoding

```
GCSSTR <Xt>, [<Xn|SP>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_GCS) then UNDEFINED;
integer n = UInt(Rn);
integer t = UInt(Rt);
```

Assembler symbols

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

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

Operation

```
bits(64) address;
bits(64) data;

bits(2) effective_el = PSTATE.EL;

if effective_el == PSTATE.EL then
    CheckGCSSTREnabled();

AccessDescriptor accdesc = CreateAccDescGCS(effective_el, MemOp_STORE);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

data = X[t, 64];
Mem[address, 8, accdesc] = data;
```

C6.2.140 GCSSTTR

Guarded Control Stack unprivileged Store stores a doubleword from a register to memory. The address that is used for the store is calculated from a base register.

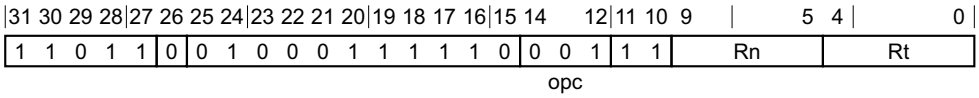
Memory accesses made by the instruction behave as if the instruction was executed at EL0 if the *Effective value* of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1 and HCR_EL2.{NV, NV1} is not {1, 1}.
- The instruction is executed at EL2 when the *Effective value* of HCR_EL2.{E2H, TGE} is {1, 1}.

Otherwise, the memory access operates with the restrictions determined by the Exception level at which the instruction is executed.

Integer

(FEAT_GCS)



Encoding

```
GCSSTTR <Xt>, [<Xn|SP>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_GCS) then UNDEFINED;
integer n = UInt(Rn);
integer t = UInt(Rt);
```

Assembler symbols

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

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

Operation

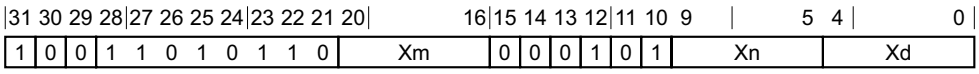
```
bits(64) address;
bits(64) data;

bits(2) effective_el = if AArch64.IsUnprivAccessPriv() then PSTATE.EL else EL0;

if effective_el == PSTATE.EL then
    CheckGCSSTREnabled();

AccessDescriptor accdesc = CreateAccDescGCS(effective_el, MemOp_STORE);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

data = X[t, 64];
Mem[address, 8, accdesc] = data;
```

C6.2.141 GMI

Tag Mask Insert inserts the tag in the first source register into the excluded set specified in the second source register, writing the new excluded set to the destination register.

Integer

(FEAT_MTE)



Encoding

```
GMI <Xd>, <Xn|SP>, <Xm>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_MTE) then UNDEFINED;
integer d = UInt(Xd);
integer n = UInt(Xn);
integer m = UInt(Xm);
```

Assembler symbols

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

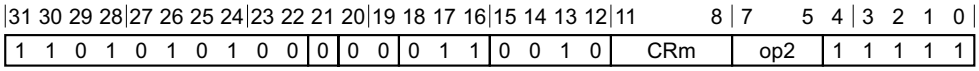
Operation

```
bits(64) address = if n == 31 then SP[] else X[n, 64];
bits(64) mask = X[m, 64];
bits(4) tag = AArch64.AllocationTagFromAddress(address);

mask<UInt(tag)> = '1';
X[d, 64] = mask;
```

C6.2.142 HINT

Hint instruction is for the instruction set space that is reserved for architectural hint instructions.

Some encodings described here are not allocated in this revision of the architecture, and behave as NOPs. These encodings might be allocated to other hint functionality in future revisions of the architecture and therefore must not be used by software.



Encoding

HINT #<imm>

Decode for this encoding

```
SystemHintOp op;

case CRm:op2 of
    when '0000 000' op = SystemHintOp_NOP;
    when '0000 001' op = SystemHintOp_YIELD;
    when '0000 010' op = SystemHintOp_WFE;
    when '0000 011' op = SystemHintOp_WFI;
    when '0000 100' op = SystemHintOp_SEV;
    when '0000 101' op = SystemHintOp_SEVL;
    when '0000 110'
        if !IsFeatureImplemented(FEAT_DGH) then EndOfInstruction();    // Instruction executes as NOP
        op = SystemHintOp_DGH;
    when '0000 111' SEE "XPACLRI";
    when '0001 xxx'
        case op2 of
            when '000' SEE "PACIA1716";
            when '010' SEE "PACIB1716";
            when '100' SEE "AUTIA1716";
            when '110' SEE "AUTIB1716";
            otherwise EndOfInstruction();
    when '0010 000'
        if !IsFeatureImplemented(FEAT_RAS) then EndOfInstruction();    // Instruction executes as NOP
        op = SystemHintOp_ESB;
    when '0010 001'
        if !IsFeatureImplemented(FEAT_SPE) then EndOfInstruction();    // Instruction executes as NOP
        op = SystemHintOp_PSB;
    when '0010 010'
        if !IsFeatureImplemented(FEAT_TRF) then EndOfInstruction();    // Instruction executes as NOP
        op = SystemHintOp_TSB;
    when '0010 011'
        if !IsFeatureImplemented(FEAT_GCS) then EndOfInstruction();    // Instruction executes as NOP
        op = SystemHintOp_GCSB;
    when '0010 100'
        op = SystemHintOp_CSDB;
    when '0010 110'
        if !IsFeatureImplemented(FEAT_CLRBHB) then
            EndOfInstruction();
        op = SystemHintOp_CLRBHB;
    when '0011 xxx'
        case op2 of
            when '000' SEE "PACIAZ";
            when '001' SEE "PACIASP";
            when '010' SEE "PACIBZ";
            when '011' SEE "PACIBSP";
            when '100' SEE "AUTIAZ";
```

```
            when '101' SEE "AUTIASP";
            when '110' SEE "AUTIBZ";
            when '111' SEE "AUTIBSP";
    when '0100 xx0'
        op = SystemHintOp_BTI;
        // Check branch target compatibility between BTI instruction and PSTATE.BTYPE
        SetBTypeCompatible(BTypeCompatible_BTI(op2<2:1>));
    when '0101 000'
        if !IsFeatureImplemented(FEAT_CHK) then EndOfInstruction();    // Instruction executes as NOP
        op = SystemHintOp_CHKFEAT;
    otherwise EndOfInstruction();
```

Assembler symbols

<imm> Is a 7-bit unsigned immediate, in the range 0 to 127, encoded in the "CRm:op2" field.

The encodings that are allocated to architectural hint functionality are described in the 'Hints' table in the 'Index by Encoding'.

Note

For allocated encodings of "CRm:op2":

- A disassembler will disassemble the allocated instruction, rather than the HINT instruction.
- An assembler may support assembly of allocated encodings using HINT with the corresponding <imm> value, but it is not required to do so.

Operation

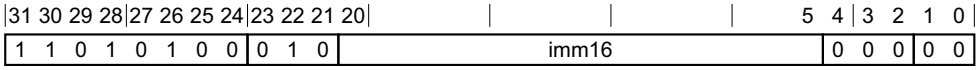
```
case op of
    when SystemHintOp_YIELD
        Hint_Yield();

    when SystemHintOp_DGH
        Hint_DGH();

    when SystemHintOp_WFE
        integer localtimeout = 1 << 64;    // No local timeout event is generated
        Hint_WFE(localtimeout, WFxType_WFE);

    when SystemHintOp_WFI
        integer localtimeout = 1 << 64;    // No local timeout event is generated
        Hint_WFI(localtimeout, WFxType_WFI);

    when SystemHintOp_SEV
        SendEvent();

    when SystemHintOp_SEVL
        SendEventLocal();

    when SystemHintOp_ESB
        if IsFeatureImplemented(FEAT_TME) && TSTATE.depth > 0 then
            FailTransaction(TMFailure_ERR, FALSE);
        SynchronizeErrors();
        AArch64.ESBOperation();
        if PSTATE.EL IN {EL0, EL1} && EL2Enabled() then AArch64.vESBOperation();
        TakeUnmaskedSErrorInterrupts();

    when SystemHintOp_PSB
        ProfilingSynchronizationBarrier();

    when SystemHintOp_TSB
        TraceSynchronizationBarrier();

    when SystemHintOp_GCSB
```

```
    GCSSynchronizationBarrier();

when SystemHintOp_CHKFEAT
    X[16, 64] = AArch64.ChkFeat(X[16, 64]);

when SystemHintOp_CSDB
    ConsumptionOfSpeculativeDataBarrier();

when SystemHintOp_CLRBHB
    Hint_CLRBHB();

when SystemHintOp_BTI
    SetBTypeNext('00');

when SystemHintOp_NOP
    return;    // do nothing

otherwise
    Unreachable();
```

C6.2.143 HLT

Halt instruction. An HLT instruction can generate a Halt Instruction debug event, which causes entry into Debug state.

Within a guarded memory region, while PSTATE.BTYPE != 0b00, a HLT instruction that would cause entry into Debug state will not generate a Branch Target Exception and will cause entry into Debug state as normal. For more information, see *PSTATE.BTYPE*.



Encoding

```
HLT #<imm>
```

Decode for this encoding

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

Assembler symbols

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

Operation

```
FaultRecord fault = NoFault();
Halt(DebugHalt_HaltInstruction, FALSE, fault);
```

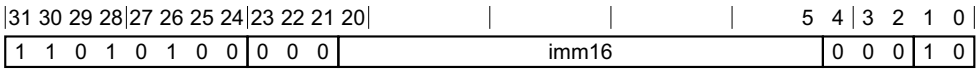
C6.2.144 HVC

Hypervisor Call causes an exception to EL2. 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 SCR_EL3.HCE is set to 0.
- When EL3 is not implemented and HCR_EL2.HCD is set to 1.
- When EL2 is not implemented.
- At EL1 if EL2 is not enabled in the current Security state.
- At EL0.

On executing an HVC instruction, the PE records the exception as a Hypervisor Call exception in ESR_ELx, using the EC value 0x16, and the value of the immediate argument.



Encoding

HVC #<imm>

Decode for this encoding

```
// Empty.
```

Assembler symbols

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

Operation

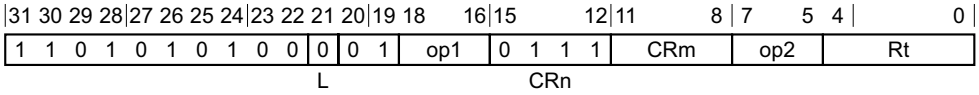
```
if !HaveEL(EL2) || PSTATE.EL == EL0 || (PSTATE.EL == EL1 && !EL2Enabled()) then
    UNDEFINED;

bits(1) hvc_enable = if HaveEL(EL3) then SCR_EL3.HCE else NOT(HCR_EL2.HCD);

if hvc_enable == '0' then
    UNDEFINED;
else
    AArch64.CallHypervisor(imm16);
```

C6.2.145 IC

Instruction Cache operation. For more information, see *op0==0b01, cache maintenance, TLB maintenance, address translation, prediction restriction, BRBE, Trace Extension, and Guarded Control Stack instructions*.

This instruction is an alias of the SYS instruction. This means that:

- The encodings in this description are named to match the encodings of SYS.
- The description of SYS gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.



Encoding

```
IC <ic_op>{, <Xt>}
```

is equivalent to

```
SYS #<op1>, C7, <Cm>, #<op2>{, <Xt>}
```

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

Assembler symbols

`<ic_op>` Is an IC instruction name, as listed for the IC system instruction pages, encoded in the "op1:CRm:op2" field. It can have the following values:

`IALLUIS`	when `op1 = 000`, `CRm = 0001`, `op2 = 000`
`IALLU`	when `op1 = 000`, `CRm = 0101`, `op2 = 000`
`IVAU`	when `op1 = 011`, `CRm = 0101`, `op2 = 001`

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

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

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

`<Xt>` Is the 64-bit name of the optional general-purpose source register, defaulting to '11111', encoded in the "Rt" field.

Operation

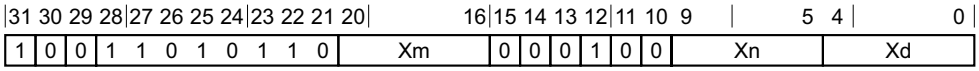
The description of SYS gives the operational pseudocode for this instruction.

C6.2.146 IRG

Insert Random Tag inserts a random Logical Address Tag into the address in the first source register, and writes the result to the destination register. Any tags specified in the optional second source register or in GCR_EL1.Exclude are excluded from the selection of the random Logical Address Tag.

Integer

(FEAT_MTE)



Encoding

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

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_MTE) then UNDEFINED;
integer d = UInt(Xd);
integer n = UInt(Xn);
integer m = UInt(Xm);
```

Assembler symbols

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

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

<Xm> Is the 64-bit name of the second general-purpose source register, encoded in the "Xm" field. Defaults to XZR if absent.

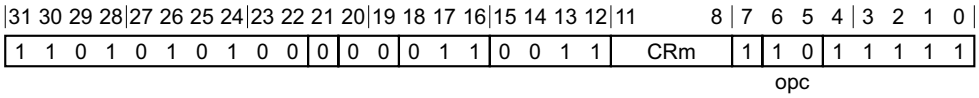
Operation

```
bits(64) operand = if n == 31 then SP[] else X[n, 64];
bits(64) exclude_reg = X[m, 64];
bits(16) exclude = exclude_reg<15:0> OR GCR_EL1.Exclude;
bits(4) rtag;

if AArch64.AllocationTagAccessIsEnabled(PSTATE.EL) then
    if GCR_EL1.RRND == '1' then
        if IsOnes(exclude) then
            rtag = '0000';
        else
            rtag = ChooseRandomNonExcludedTag(exclude);
    else
        bits(4) start_tag = RGSR_EL1.TAG;
        bits(4) offset = AArch64.RandomTag();

        rtag = AArch64.ChooseNonExcludedTag(start_tag, offset, exclude);

        RGSR_EL1.TAG = rtag;
else
    rtag = '0000';

bits(64) result = AArch64.AddressWithAllocationTag(operand, rtag);

if d == 31 then
```

```
    SP[] = result;
else
    X[d, 64] = result;
```

C6.2.147 ISB

Instruction Synchronization Barrier flushes the pipeline in the PE and is a context synchronization event. For more information, see *Instruction Synchronization Barrier (ISB)*.



Encoding

```
ISB {<option>|#<imm>}
```

Decode for this encoding

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

Assembler symbols

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

SY Full system barrier operation, encoded as CRm = 0b1111. Can be omitted.

All other encodings of "CRm" are reserved. The corresponding instructions execute as full system barrier operations, but must not be relied upon by software.

<imm> Is an optional 4-bit unsigned immediate, in the range 0 to 15, defaulting to 15 and encoded in the "CRm" field.

Operation

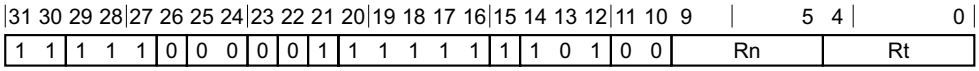
```
InstructionSynchronizationBarrier();
if IsFeatureImplemented(FEAT_BRBE) && BRBEBranchOnISB() then
    BRBEISB();
```

C6.2.148 LD64B

Single-copy Atomic 64-byte Load derives an address from a base register value, loads eight 64-bit doublewords from a memory location, and writes them to consecutive registers, `Xt` to `X(t+7)`. The data that is loaded is atomic and is required to be 64-byte aligned.

Integer

(FEAT_LS64)



Encoding

```
LD64B <Xt>, [<Xn|SP> {,#0}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_LS64) then UNDEFINED;
if Rt<4:3> == '11' || Rt<0> == '1' then UNDEFINED;

integer n = UInt(Rn);
integer t = UInt(Rt);
MemOp memop = MemOp_LOAD;
boolean tagchecked = n != 31;
```

Assembler symbols

<Xt>	Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
CheckLDST64BEnabled();

bits(512) data;
bits(64) address;
bits(64) value;

AccessDescriptor accdesc = CreateAccDescLS64(memop, tagchecked);
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

data = MemLoad64B(address, accdesc);

for i = 0 to 7
    value = data<63+64*i:64*i>;
    if BigEndian(accdesc.acctype) then value = BigEndianReverse(value);
    X[t+i, 64] = value;
```

C6.2.149 LDADD, LDADDA, LDADDAL, LDADDL

Atomic add on word or doubleword in memory atomically loads a 32-bit word or 64-bit doubleword from memory, adds the value held in a register to it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not one of `WZR` or `XZR`, `LDADDA` and `LDADDAL` load from memory with acquire semantics.
- `LDADDL` and `LDADDAL` store to memory with release semantics.
- `LDADD` has neither acquire nor release semantics.

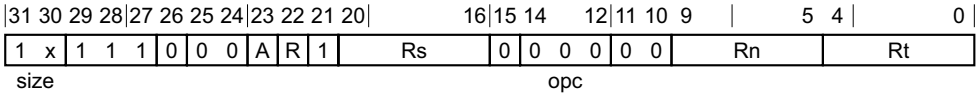
For more information about memory ordering semantics, see *Load-Acquire, Load-AcquirePC, and Store-Release*.

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

This instruction is used by the alias STADD, STADDL. See *Alias conditions* for details of when each alias is preferred.

Integer

(FEAT_LSE)



32-bit LDADD variant

Applies when `size == 10 && A == 0 && R == 0`.

```
LDADD <Ws>, <Wt>, [<Xn|SP>]
```

32-bit LDADDA variant

Applies when `size == 10 && A == 1 && R == 0`.

```
LDADDA <Ws>, <Wt>, [<Xn|SP>]
```

32-bit LDADDAL variant

Applies when `size == 10 && A == 1 && R == 1`.

```
LDADDAL <Ws>, <Wt>, [<Xn|SP>]
```

32-bit LDADDL variant

Applies when `size == 10 && A == 0 && R == 1`.

```
LDADDL <Ws>, <Wt>, [<Xn|SP>]
```

64-bit LDADD variant

Applies when `size == 11 && A == 0 && R == 0`.

```
LDADD <Xs>, <Xt>, [<Xn|SP>]
```

64-bit LDADDA variant

Applies when `size == 11 && A == 1 && R == 0`.

```
LDADDA <Xs>, <Xt>, [<Xn|SP>]
```

64-bit LDADDAL variant

Applies when size == 11 && A == 1 && R == 1.

```
LDADDAL <Xs>, <Xt>, [<Xn|SP>]
```

64-bit LDADDL variant

Applies when size == 11 && A == 0 && R == 1.

```
LDADDL <Xs>, <Xt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LSE) then UNDEFINED;

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

constant integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
boolean acquire = A == '1' && Rt != '11111';
boolean release = R == '1';
boolean tagchecked = n != 31;
```

Alias conditions

Alias	is preferred when
STADD, STADDL	A == '0' && Rt == '11111'

Assembler symbols

<Ws>	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Wt>	Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xs>	Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Xt>	Is the 64-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(datasize) value;
bits(datasize) data;

AccessDescriptor accdesc = CreateAccDescAtomicOp(MemAtomicOp_ADD, acquire, release, tagchecked);

value = X[s, datasize];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

bits(datasize) comparevalue = bits(datasize) UNKNOWN;    // Irrelevant when not executing CAS
data = MemAtomic(address, comparevalue, value, accdesc);
```

```
if t != 31 then
    X[t, regsize] = ZeroExtend(data, regsize);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

C6.2.150 LDADDB, LDADDAB, LDADDALB, LDADDLB

Atomic add on byte in memory atomically loads an 8-bit byte from memory, adds the value held in a register to it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, LDADDAB and LDADDALB load from memory with acquire semantics.
- LDADDLB and LDADDALB store to memory with release semantics.
- LDADDB has neither acquire nor release semantics.

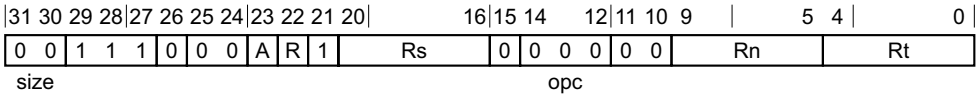
For more information about memory ordering semantics, see *Load-Acquire, Load-AcquirePC, and Store-Release*.

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

This instruction is used by the alias STADDB, STADDLB. See *Alias conditions* for details of when each alias is preferred.

Integer

(FEAT_LSE)



LDADDAB variant

Applies when A == 1 && R == 0.

```
LDADDAB <Ws>, <Wt>, [<Xn|SP>]
```

LDADDALB variant

Applies when A == 1 && R == 1.

```
LDADDALB <Ws>, <Wt>, [<Xn|SP>]
```

LDADDB variant

Applies when A == 0 && R == 0.

```
LDADDB <Ws>, <Wt>, [<Xn|SP>]
```

LDADDLB variant

Applies when A == 0 && R == 1.

```
LDADDLB <Ws>, <Wt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LSE) then UNDEFINED;

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

boolean acquire = A == '1' && Rt != '11111';
boolean release = R == '1';
boolean tagchecked = n != 31;
```

Alias conditions

Alias	is preferred when
STADDB, STADDLB	`A == '0' && Rt == '11111'`

Assembler symbols

`<Ws>` Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

`<Wt>` Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

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

Operation

```
bits(64) address;
bits(8) value;
bits(8) data;

AccessDescriptor accdesc = CreateAccDescAtomicOp(MemAtomicOp_ADD, acquire, release, tagchecked);

value = X[s, 8];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

bits(8) comparevalue = bits(8) UNKNOWN;    // Irrelevant when not executing CAS
data = MemAtomic(address, comparevalue, value, accdesc);

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

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

C6.2.151 LDADDH, LDADDAH, LDADDALH, LDADDLH

Atomic add on halfword in memory atomically loads a 16-bit halfword from memory, adds the value held in a register to it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, LDADDAH and LDADDALH load from memory with acquire semantics.
- LDADDLH and LDADDALH store to memory with release semantics.
- LDADDH has neither acquire nor release semantics.

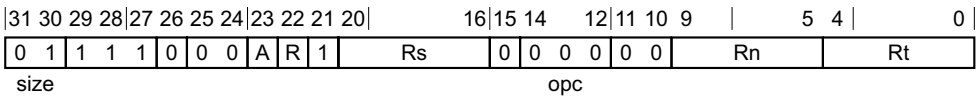
For more information about memory ordering semantics, see *Load-Acquire, Load-AcquirePC, and Store-Release*.

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

This instruction is used by the alias STADDH, STADDLH. See *Alias conditions* for details of when each alias is preferred.

Integer

(FEAT_LSE)



LDADDAH variant

Applies when A == 1 && R == 0.

```
LDADDAH <Ws>, <Wt>, [<Xn|SP>]
```

LDADDALH variant

Applies when A == 1 && R == 1.

```
LDADDALH <Ws>, <Wt>, [<Xn|SP>]
```

LDADDH variant

Applies when A == 0 && R == 0.

```
LDADDH <Ws>, <Wt>, [<Xn|SP>]
```

LDADDLH variant

Applies when A == 0 && R == 1.

```
LDADDLH <Ws>, <Wt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LSE) then UNDEFINED;

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

boolean acquire = A == '1' && Rt != '11111';
boolean release = R == '1';
boolean tagchecked = n != 31;
```

Alias conditions

Alias	is preferred when
STADDH, STADDLH	`A == '0' && Rt == '11111'`

Assembler symbols

`<Ws>` Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

`<Wt>` Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

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

Operation

```
bits(64) address;
bits(16) value;
bits(16) data;

AccessDescriptor accdesc = CreateAccDescAtomicOp(MemAtomicOp_ADD, acquire, release, tagchecked);

value = X[s, 16];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

bits(16) comparevalue = bits(16) UNKNOWN;    // Irrelevant when not executing CAS
data = MemAtomic(address, comparevalue, value, accdesc);

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

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

C6.2.152 LDAPR

Load-Acquire RCpc Register derives an address from a base register value, loads a 32-bit word or 64-bit doubleword from the derived address in memory, and writes it to a register.

The instruction has memory ordering semantics as described in *Load-Acquire, Load-AcquirePC, and Store-Release*, except that:

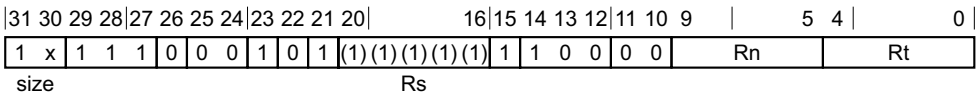
- There is no ordering requirement, separate from the requirements of a Load-AcquirePC or a Store-Release, created by having a Store-Release followed by a Load-AcquirePC instruction.
- The reading of a value written by a Store-Release by a Load-AcquirePC instruction by the same observer does not make the write of the Store-Release globally observed.

This difference in memory ordering is not described in the pseudocode.

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

No offset

(FEAT_LRCPC)



32-bit variant

Applies when `size == 10`.

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

64-bit variant

Applies when `size == 11`.

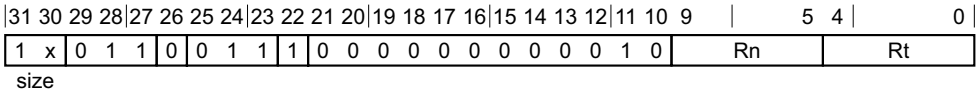
```
LDAPR <Xt>, [<Xn|SP> {,#0}]
```

Decode for all variants of this encoding

```
boolean wback = FALSE;
integer offset = 0;
boolean wb_unknown = FALSE;

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

constant integer elsize = 8 << UInt(size);
constant integer regsize = if elsize == 64 then 64 else 32;
constant integer datasize = elsize;
boolean tagchecked = n != 31;
```

Post-index

(FEAT_LRCPC3)



32-bit variant

Applies when size == 10.

```
LDAPR <Wt>, [<Xn|SP>], #4
```

64-bit variant

Applies when size == 11.

```
LDAPR <Xt>, [<Xn|SP>], #8
```

Decode for all variants of this encoding

```
boolean wback = TRUE;

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

constant integer regsize = if size == '11' then 64 else 32;
constant integer datasize = 8 << UInt(size);
constant integer offset = 1 << UInt(size);

boolean tagchecked = TRUE;

boolean wb_unknown = FALSE;

if n == t && n != 31 then
    Constraint c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;    // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;    // writeback is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();
```

Assembler symbols

<Wt>	Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xt>	Is the 64-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation for all encodings

```
bits(64) address;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;

AccessDescriptor accdesc = CreateAccDescLDAcqPC(tagchecked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

data = Mem[address, dbytes, accdesc];
X[t, regsize] = ZeroExtend(data, regsize);

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    else
```

```
    address = GenerateAddress(address, offset, accdesc);
if n == 31 then
    SP[] = address;
else
    X[n, 64] = address;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

C6.2.153 LDAPRB

Load-Acquire RCpc Register Byte derives an address from a base register value, loads a byte from the derived address in memory, zero-extends it and writes it to a register.

The instruction has memory ordering semantics as described in *Load-Acquire, Load-AcquirePC, and Store-Release*, except that:

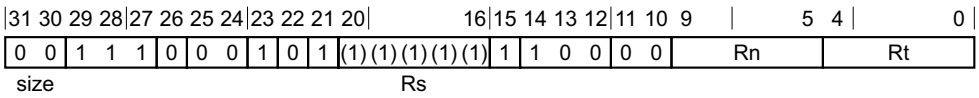
- There is no ordering requirement, separate from the requirements of a Load-AcquirePC or a Store-Release, created by having a Store-Release followed by a Load-AcquirePC instruction.
- The reading of a value written by a Store-Release by a Load-AcquirePC instruction by the same observer does not make the write of the Store-Release globally observed.

This difference in memory ordering is not described in the pseudocode.

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

Integer

(FEAT_LRCPC)



Encoding

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

Decode for this encoding

```
integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tagchecked = n != 31;
```

Assembler symbols

<Wt>	Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(8) data;

AccessDescriptor accdesc = CreateAccDescLDAcqPC(tagchecked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

data = Mem[address, 1, accdesc];
X[t, 32] = ZeroExtend(data, 32);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

C6.2.154 LDAPRH

Load-Acquire RCpc Register Halfword derives an address from a base register value, loads a halfword from the derived address in memory, zero-extends it and writes it to a register.

The instruction has memory ordering semantics as described in *Load-Acquire, Load-AcquirePC, and Store-Release*, except that:

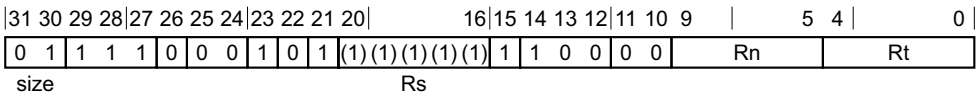
- There is no ordering requirement, separate from the requirements of a Load-AcquirePC or a Store-Release, created by having a Store-Release followed by a Load-AcquirePC instruction.
- The reading of a value written by a Store-Release by a Load-AcquirePC instruction by the same observer does not make the write of the Store-Release globally observed.

This difference in memory ordering is not described in the pseudocode.

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

Integer

(FEAT_LRCPC)



Encoding

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

Decode for this encoding

```
integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tagchecked = n != 31;
```

Assembler symbols

<Wt>	Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(16) data;

AccessDescriptor accdesc = CreateAccDescLDAcqPC(tagchecked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

data = Mem[address, 2, accdesc];
X[t, 32] = ZeroExtend(data, 32);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

C6.2.155 LDAPUR

Load-Acquire RCpc Register (unscaled) calculates an address from a base register and an immediate offset, loads a 32-bit word or 64-bit doubleword from memory, zero-extends it, and writes it to a register.

The instruction has memory ordering semantics as described in *Load-Acquire, Load-AcquirePC, and Store-Release*, except that:

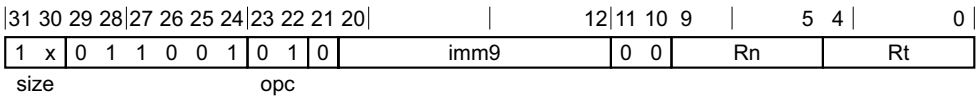
- There is no ordering requirement, separate from the requirements of a Load-AcquirePC or a Store-Release, created by having a Store-Release followed by a Load-AcquirePC instruction.
- The reading of a value written by a Store-Release by a Load-AcquirePC instruction by the same observer does not make the write of the Store-Release globally observed.

This difference in memory ordering is not described in the pseudocode.

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

Unscaled offset

(FEAT_LRCPC2)



32-bit variant

Applies when `size == 10`.

```
LDAPUR <Wt>, [<Xn|SP>{, #<simm>}]
```

64-bit variant

Applies when `size == 11`.

```
LDAPUR <Xt>, [<Xn|SP>{, #<simm>}]
```

Decode for all variants of this encoding

```
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

Assembler symbols

`<Wt>`	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
`<Xt>`	Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
`<Xn|SP>`	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
`<simm>`	Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared decode for all encodings

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer regsize;

regsize = if size == '11' then 64 else 32;
```

```
constant integer datasize = 8 << scale;
boolean tagchecked = n != 31;
```

Operation

```
bits(64) address;
bits(datasize) data;

AccessDescriptor accdesc;
accdesc = CreateAccDescLDAcqPC(tagchecked);
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

address = GenerateAddress(address, offset, accdesc);

data = Mem[address, datasize DIV 8, accdesc];
X[t, regsize] = ZeroExtend(data, regsize);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

C6.2.156 LDAPURB

Load-Acquire RCpc Register Byte (unscaled) calculates an address from a base register and an immediate offset, loads a byte from memory, zero-extends it, and writes it to a register.

The instruction has memory ordering semantics as described in *Load-Acquire, Load-AcquirePC, and Store-Release*, except that:

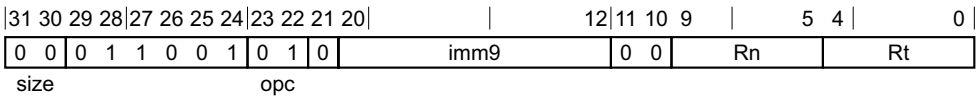
- There is no ordering requirement, separate from the requirements of a Load-AcquirePC or a Store-Release, created by having a Store-Release followed by a Load-AcquirePC instruction.
- The reading of a value written by a Store-Release by a Load-AcquirePC instruction by the same observer does not make the write of the Store-Release globally observed.

This difference in memory ordering is not described in the pseudocode.

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

Unscaled offset

(FEAT_LRCPC2)



Encoding

```
LDAPURB <Wt>, [<Xn|SP>{, #<simm>}]
```

Decode for this encoding

```
bits(64) offset = SignExtend(imm9, 64);
```

Assembler symbols

<Wt>	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<simm>	Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared decode for all encodings

```
integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tagchecked = n != 31;
```

Operation

```
bits(64) address;
bits(8) data;

AccessDescriptor accdesc;
accdesc = CreateAccDescLDAcqPC(tagchecked);
if n == 31 then
    CheckSPAlignment();
    address = SP[];
```

```
else
    address = X[n, 64];

address = GenerateAddress(address, offset, accdesc);

data = Mem[address, 1, accdesc];
X[t, 32] = ZeroExtend(data, 32);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

C6.2.157 LDAPURH

Load-Acquire RCpc Register Halfword (unscaled) calculates an address from a base register and an immediate offset, loads a halfword from memory, zero-extends it, and writes it to a register.

The instruction has memory ordering semantics as described in *Load-Acquire, Load-AcquirePC, and Store-Release*, except that:

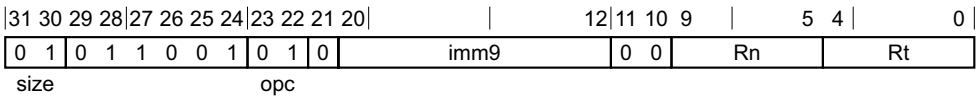
- There is no ordering requirement, separate from the requirements of a Load-AcquirePC or a Store-Release, created by having a Store-Release followed by a Load-AcquirePC instruction.
- The reading of a value written by a Store-Release by a Load-AcquirePC instruction by the same observer does not make the write of the Store-Release globally observed.

This difference in memory ordering is not described in the pseudocode.

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

Unscaled offset

(FEAT_LRCPC2)



Encoding

```
LDAPURH <Wt>, [<Xn|SP>{, #<simm>}]
```

Decode for this encoding

```
bits(64) offset = SignExtend(imm9, 64);
```

Assembler symbols

<Wt>	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<simm>	Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared decode for all encodings

```
integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tagchecked = n != 31;
```

Operation

```
bits(64) address;
bits(16) data;

AccessDescriptor accdesc;
accdesc = CreateAccDescLDAcqPC(tagchecked);
if n == 31 then
    CheckSPAlignment();
    address = SP[];
```

```
else
    address = X[n, 64];

address = GenerateAddress(address, offset, accdesc);

data = Mem[address, 2, accdesc];
X[t, 32] = ZeroExtend(data, 32);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

C6.2.158 LDAPURSB

Load-Acquire RCpc Register Signed Byte (unscaled) calculates an address from a base register and an immediate offset, loads a signed byte from memory, sign-extends it, and writes it to a register.

The instruction has memory ordering semantics as described in *Load-Acquire, Load-AcquirePC, and Store-Release*, except that:

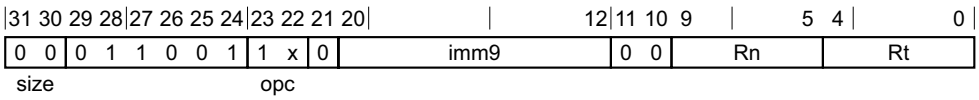
- There is no ordering requirement, separate from the requirements of a Load-AcquirePC or a Store-Release, created by having a Store-Release followed by a Load-AcquirePC instruction.
- The reading of a value written by a Store-Release by a Load-AcquirePC instruction by the same observer does not make the write of the Store-Release globally observed.

This difference in memory ordering is not described in the pseudocode.

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

Unscaled offset

(FEAT_LRCPC2)



32-bit variant

Applies when `opc == 11`.

```
LDAPURSB <Wt>, [<Xn|SP>{, #<simm>}]
```

64-bit variant

Applies when `opc == 10`.

```
LDAPURSB <Xt>, [<Xn|SP>{, #<simm>}]
```

Decode for all variants of this encoding

```
bits(64) offset = SignExtend(imm9, 64);
```

Assembler symbols

`<Wt>`	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
`<Xt>`	Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
`<Xn|SP>`	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
`<simm>`	Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared decode for all encodings

```
integer n = UInt(Rn);
integer t = UInt(Rt);
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
```

```
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = 32;
    signed = FALSE;
else
    // sign-extending load
    memop = MemOp_LOAD;
    regsize = if opc<0> == '1' then 32 else 64;
    signed = TRUE;

boolean tagchecked = memop != MemOp_PREFETCH && (n != 31);
```

Operation

```
bits(64) address;
bits(8) data;

AccessDescriptor accdesc;
if memop == MemOp_LOAD then
    accdesc = CreateAccDescLDAcqPC(tagchecked);
elsif memop == MemOp_STORE then
    accdesc = CreateAccDescAcqRel(memop, tagchecked);

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

address = GenerateAddress(address, offset, accdesc);

case memop of
    when MemOp_STORE
        data = X[t, 8];
        Mem[address, 1, accdesc] = data;

    when MemOp_LOAD
        data = Mem[address, 1, accdesc];
        if signed then
            X[t, regsize] = SignExtend(data, regsize);
        else
            X[t, regsize] = ZeroExtend(data, regsize);

    when MemOp_PREFETCH
        Prefetch(address, t<4:0>);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

C6.2.159 LDAPURSH

Load-Acquire RCpc Register Signed Halfword (unscaled) calculates an address from a base register and an immediate offset, loads a signed halfword from memory, sign-extends it, and writes it to a register.

The instruction has memory ordering semantics as described in *Load-Acquire, Load-AcquirePC, and Store-Release*, except that:

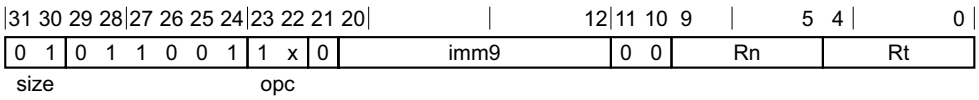
- There is no ordering requirement, separate from the requirements of a Load-AcquirePC or a Store-Release, created by having a Store-Release followed by a Load-AcquirePC instruction.
- The reading of a value written by a Store-Release by a Load-AcquirePC instruction by the same observer does not make the write of the Store-Release globally observed.

This difference in memory ordering is not described in the pseudocode.

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

Unscaled offset

(FEAT_LRCPC2)



32-bit variant

Applies when `opc == 11`.

```
LDAPURSH <Wt>, [<Xn|SP>{, #<simm>}]
```

64-bit variant

Applies when `opc == 10`.

```
LDAPURSH <Xt>, [<Xn|SP>{, #<simm>}]
```

Decode for all variants of this encoding

```
bits(64) offset = SignExtend(imm9, 64);
```

Assembler symbols

`<Wt>`	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
`<Xt>`	Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
`<Xn|SP>`	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
`<simm>`	Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared decode for all encodings

```
integer n = UInt(Rn);
integer t = UInt(Rt);
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
```

```
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = 32;
    signed = FALSE;
else
    // sign-extending load
    memop = MemOp_LOAD;
    regsize = if opc<0> == '1' then 32 else 64;
    signed = TRUE;

boolean tagchecked = memop != MemOp_PREFETCH && (n != 31);
```

Operation

```
bits(64) address;
bits(16) data;

AccessDescriptor accdesc;
if memop == MemOp_LOAD then
    accdesc = CreateAccDescLDAcqPC(tagchecked);
elsif memop == MemOp_STORE then
    accdesc = CreateAccDescAcqRel(memop, tagchecked);

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

address = GenerateAddress(address, offset, accdesc);

case memop of
    when MemOp_STORE
        data = X[t, 16];
        Mem[address, 2, accdesc] = data;

    when MemOp_LOAD
        data = Mem[address, 2, accdesc];
        if signed then
            X[t, regsize] = SignExtend(data, regsize);
        else
            X[t, regsize] = ZeroExtend(data, regsize);

    when MemOp_PREFETCH
        Prefetch(address, t<4:0>);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

C6.2.160 LDAPURSW

Load-Acquire RCpc Register Signed Word (unscaled) calculates an address from a base register and an immediate offset, loads a signed word from memory, sign-extends it, and writes it to a register.

The instruction has memory ordering semantics as described in *Load-Acquire, Load-AcquirePC, and Store-Release*, except that:

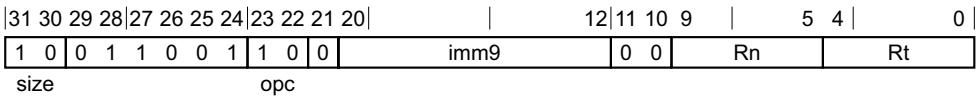
- There is no ordering requirement, separate from the requirements of a Load-AcquirePC or a Store-Release, created by having a Store-Release followed by a Load-AcquirePC instruction.
- The reading of a value written by a Store-Release by a Load-AcquirePC instruction by the same observer does not make the write of the Store-Release globally observed.

This difference in memory ordering is not described in the pseudocode.

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

Unscaled offset

(FEAT_LRCPC2)



Encoding

```
LDAPURSW <Xt>, [<Xn|SP>{, #<simm>}]
```

Decode for this encoding

```
bits(64) offset = SignExtend(imm9, 64);
```

Assembler symbols

<Xt>	Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<simm>	Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared decode for all encodings

```
integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tagchecked = n != 31;
```

Operation

```
bits(64) address;
bits(32) data;

AccessDescriptor accdesc;
accdesc = CreateAccDescLDAcqPC(tagchecked);
if n == 31 then
    CheckSPAlignment();
    address = SP[];
```

```
else
    address = X[n, 64];

address = GenerateAddress(address, offset, accdesc);

data = Mem[address, 4, accdesc];
X[t, 64] = SignExtend(data, 64);
```

Operational information

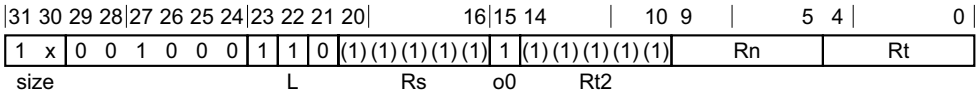
If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

C6.2.161 LDAR

Load-Acquire Register derives an address from a base register value, loads a 32-bit word or 64-bit doubleword from memory, and writes it to a register. The instruction also has memory ordering semantics as described in *Load-Acquire, Load-AcquirePC, and Store-Release*. For information about memory accesses, see *Load/store addressing modes*.

Note

For this instruction, if the destination is WZR/XZR, it is impossible for software to observe the presence of the acquire semantic other than its effect on the arrival at endpoints.



32-bit variant

Applies when `size == 10`.

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

64-bit variant

Applies when `size == 11`.

```
LDAR <Xt>, [<Xn|SP>{,#0}]
```

Decode for all variants of this encoding

```
integer n = UInt(Rn);
integer t = UInt(Rt);

constant integer elsize = 8 << UInt(size);
constant integer regsize = if elsize == 64 then 64 else 32;
boolean tagchecked = n != 31;
```

Assembler symbols

<Wt>	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xt>	Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(elsize) data;
constant integer dbytes = elsize DIV 8;

AccessDescriptor accdesc;
accdesc = CreateAccDescAcqRel(MemOp_LOAD, tagchecked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];
```

```
data = Mem[address, dbytes, accdesc];
X[t, regsize] = ZeroExtend(data, regsize);
```

Operational information

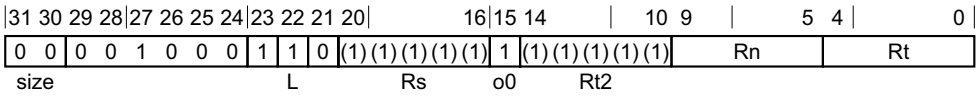
If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

C6.2.162 LDARB

Load-Acquire Register Byte derives an address from a base register value, loads a byte from memory, zero-extends it and writes it to a register. The instruction also has memory ordering semantics as described in *Load-Acquire, Load-AcquirePC, and Store-Release*. For information about memory accesses, see *Load/store addressing modes*.

Note

For this instruction, if the destination is WZR/XZR, it is impossible for software to observe the presence of the acquire semantic other than its effect on the arrival at endpoints.



Encoding

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

Decode for this encoding

```
integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tagchecked = n != 31;
```

Assembler symbols

<Wt>	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(8) data;

AccessDescriptor accdesc;
accdesc = CreateAccDescAcqRel(MemOp_LOAD, tagchecked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

data = Mem[address, 1, accdesc];
X[t, 32] = ZeroExtend(data, 32);
```

Operational information

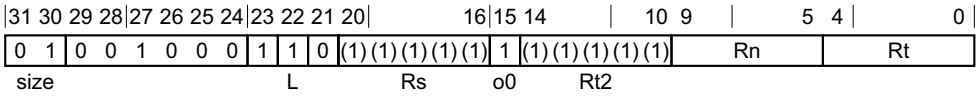
If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

C6.2.163 LDARH

Load-Acquire Register Halfword derives an address from a base register value, loads a halfword from memory, zero-extends it, and writes it to a register. The instruction also has memory ordering semantics as described in *Load-Acquire, Load-AcquirePC, and Store-Release*. For information about memory accesses, see *Load/store addressing modes*.

Note

For this instruction, if the destination is WZR/XZR, it is impossible for software to observe the presence of the acquire semantic other than its effect on the arrival at endpoints.



Encoding

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

Decode for this encoding

```
integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tagchecked = n != 31;
```

Assembler symbols

<Wt>	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

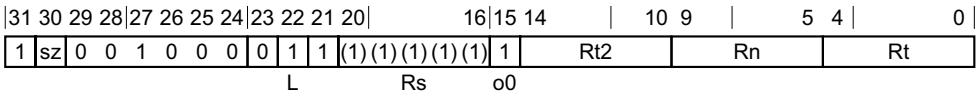
```
bits(64) address;
bits(16) data;

AccessDescriptor accdesc;
accdesc = CreateAccDescAcqRel(MemOp_LOAD, tagchecked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

data = Mem[address, 2, accdesc];
X[t, 32] = ZeroExtend(data, 32);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

C6.2.164 LDAXP

Load-Acquire Exclusive Pair of Registers derives an address from a base register value, loads two 32-bit words or two 64-bit doublewords from memory, and writes them to two registers. For information on single-copy atomicity and alignment requirements, see *Requirements for single-copy atomicity* and *Alignment of data accesses*. The PE marks the physical address being accessed as an exclusive access. This exclusive access mark is checked by Store Exclusive instructions. See *Synchronization and semaphores*. The instruction also has memory ordering semantics, as described in *Load-Acquire, Load-AcquirePC, and Store-Release*. For information about memory accesses, see *Load/store addressing modes*.



32-bit variant

Applies when sz == 0.

```
LDAXP <Wt1>, <Wt2>, [<Xn|SP>{,#0}]
```

64-bit variant

Applies when sz == 1.

```
LDAXP <Xt1>, <Xt2>, [<Xn|SP>{,#0}]
```

Decode for all variants of this encoding

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2);

constant integer elsize = 32 << UInt(sz);
constant integer datasize = elsize * 2;
boolean tagchecked = n != 31;

boolean rt_unknown = FALSE;
if t == t2 then
    Constraint c = ConstrainUnpredictable(Unpredictable_LDPOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN rt_unknown = TRUE;    // result is UNKNOWN
        when Constraint_UNDEF   UNDEFINED;
        when Constraint_NOP     EndOfInstruction();
```

Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*, and particularly *LDAXP*.

Assembler symbols

<Wt1> Is the 32-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.

<Wt2> Is the 32-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.

<Xt1> Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.

<Xt2> Is the 64-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.

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

Operation

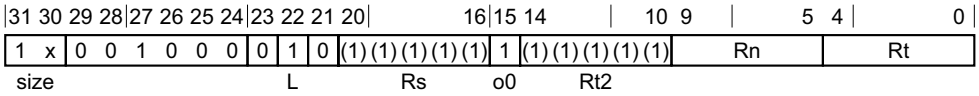
```
bits(64) address;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;

AccessDescriptor accdesc = CreateAccDescExLDST(MemOp_LOAD, TRUE, tagchecked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

// Tell the Exclusives monitors to record a sequence of one or more atomic
// memory reads from virtual address range [address, address+dbytes-1].
// The Exclusives monitor will only be set if all the reads are from the
// same dbytes-aligned physical address, to allow for the possibility of
// an atomicity break if the translation is changed between reads.
AArch64.SetExclusiveMonitors(address, dbytes);

if rt_unknown then
    // ConstrainedUNPREDICTABLE case
    X[t, datasize] = bits(datasize) UNKNOWN;    // In this case t = t2
elsif elsize == 32 then
    // 32-bit load exclusive pair (atomic)
    data = Mem[address, dbytes, accdesc];
    if BigEndian(accdesc.acctype) then
        X[t, datasize-elsize] = data<datasize-1:elsize>;
        X[t2, elsize] = data<elsize-1:0>;
    else
        X[t, elsize] = data<elsize-1:0>;
        X[t2, datasize-elsize] = data<datasize-1:elsize>;
else // elsize == 64
    // 64-bit load exclusive pair (not atomic), but must be 128-bit aligned
    if !IsAligned(address, dbytes) then
        AArch64.Abort(address, AlignmentFault(accdesc));

    bits(64) address2 = GenerateAddress(address, 8, accdesc);
    X[t, 64] = Mem[address, 8, accdesc];
    X[t2, 64] = Mem[address2, 8, accdesc];
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

C6.2.165 LDAXR

Load-Acquire Exclusive Register derives an address from a base register value, loads a 32-bit word or 64-bit doubleword from memory, and writes it to a register. The memory access is atomic. The PE marks the physical address being accessed as an exclusive access. This exclusive access mark is checked by Store Exclusive instructions. See *Synchronization and semaphores*. The instruction also has memory ordering semantics as described in *Load-Acquire, Load-AcquirePC, and Store-Release*. For information about memory accesses, see *Load/store addressing modes*.



32-bit variant

Applies when `size == 10`.

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

64-bit variant

Applies when `size == 11`.

```
LDAXR <Xt>, [<Xn|SP>{,#0}]
```

Decode for all variants of this encoding

```
integer n = UInt(Rn);
integer t = UInt(Rt);

constant integer elsize = 8 << UInt(size);
integer regsize = if elsize == 64 then 64 else 32;
boolean tagchecked = n != 31;
```

Assembler symbols

`<Wt>`	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
`<Xt>`	Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
`<Xn|SP>`	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(elsize) data;
constant integer dbytes = elsize DIV 8;

AccessDescriptor accdesc = CreateAccDescExLDST(MemOp_LOAD, TRUE, tagchecked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

// Tell the Exclusives monitors to record a sequence of one or more atomic
// memory reads from virtual address range [address, address+dbytes-1].
// The Exclusives monitor will only be set if all the reads are from the
// same dbytes-aligned physical address, to allow for the possibility of
```

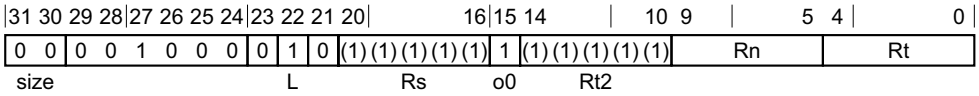
```
// an atomicity break if the translation is changed between reads.
AArch64.SetExclusiveMonitors(address, dbytes);

data = Mem[address, dbytes, accdesc];
X[t, regsize] = ZeroExtend(data, regsize);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

C6.2.166 LDAXRB

Load-Acquire Exclusive Register Byte derives an address from a base register value, loads a byte from memory, zero-extends it and writes it to a register. The memory access is atomic. The PE marks the physical address being accessed as an exclusive access. This exclusive access mark is checked by Store Exclusive instructions. See *Synchronization and semaphores*. The instruction also has memory ordering semantics as described in *Load-Acquire, Load-AcquirePC, and Store-Release*. For information about memory accesses, see *Load/store addressing modes*.



Encoding

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

Decode for this encoding

```
integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tagchecked = n != 31;
```

Assembler symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

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

Operation

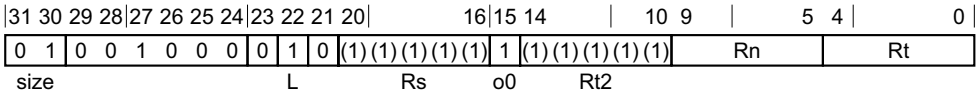
```
bits(64) address;
bits(8) data;

AccessDescriptor accdesc = CreateAccDescExLDST(MemOp_LOAD, TRUE, tagchecked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

// Tell the Exclusives monitors to record a sequence of one or more atomic
// memory reads from virtual address range [address, address+dbytes-1].
// The Exclusives monitor will only be set if all the reads are from the
// same dbytes-aligned physical address, to allow for the possibility of
// an atomicity break if the translation is changed between reads.
AArch64.SetExclusiveMonitors(address, 1);

data = Mem[address, 1, accdesc];
X[t, 32] = ZeroExtend(data, 32);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

C6.2.167 LDAXRH

Load-Acquire Exclusive Register Halfword derives an address from a base register value, loads a halfword from memory, zero-extends it and writes it to a register. The memory access is atomic. The PE marks the physical address being accessed as an exclusive access. This exclusive access mark is checked by Store Exclusive instructions. See *Synchronization and semaphores*. The instruction also has memory ordering semantics as described in *Load-Acquire, Load-AcquirePC, and Store-Release*. For information about memory accesses, see *Load/store addressing modes*.



Encoding

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

Decode for this encoding

```
integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tagchecked = n != 31;
```

Assembler symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

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

Operation

```
bits(64) address;
bits(16) data;

AccessDescriptor accdesc = CreateAccDescExLDST(MemOp_LOAD, TRUE, tagchecked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

// Tell the Exclusives monitors to record a sequence of one or more atomic
// memory reads from virtual address range [address, address+dbytes-1].
// The Exclusives monitor will only be set if all the reads are from the
// same dbytes-aligned physical address, to allow for the possibility of
// an atomicity break if the translation is changed between reads.
AArch64.SetExclusiveMonitors(address, 2);

data = Mem[address, 2, accdesc];
X[t, 32] = ZeroExtend(data, 32);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

C6.2.168 LDCLR, LDCLRA, LDCLRAL, LDCLRL

Atomic bit clear on word or doubleword in memory atomically loads a 32-bit word or 64-bit doubleword from memory, performs a bitwise AND with the complement of the value held in a register on it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not one of `WZR` or `XZR`, `LDCLRA` and `LDCLRAL` load from memory with acquire semantics.
- `LDCLRL` and `LDCLRAL` store to memory with release semantics.
- `LDCLR` has neither acquire nor release semantics.

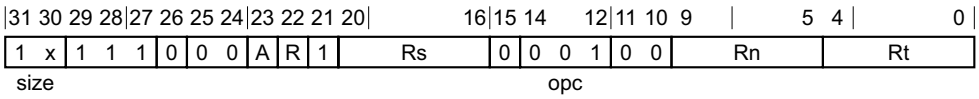
For more information about memory ordering semantics, see *Load-Acquire, Load-AcquirePC, and Store-Release*.

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

This instruction is used by the alias STCLR, STCLRL. See *Alias conditions* for details of when each alias is preferred.

Integer

(FEAT_LSE)



32-bit LDCLR variant

Applies when `size == 10 && A == 0 && R == 0`.

```
LDCLR <Ws>, <Wt>, [<Xn|SP>]
```

32-bit LDCLRA variant

Applies when `size == 10 && A == 1 && R == 0`.

```
LDCLRA <Ws>, <Wt>, [<Xn|SP>]
```

32-bit LDCLRAL variant

Applies when `size == 10 && A == 1 && R == 1`.

```
LDCLRAL <Ws>, <Wt>, [<Xn|SP>]
```

32-bit LDCLRL variant

Applies when `size == 10 && A == 0 && R == 1`.

```
LDCLRL <Ws>, <Wt>, [<Xn|SP>]
```

64-bit LDCLR variant

Applies when `size == 11 && A == 0 && R == 0`.

```
LDCLR <Xs>, <Xt>, [<Xn|SP>]
```

64-bit LDCLRA variant

Applies when `size == 11 && A == 1 && R == 0`.

```
LDCLRA <Xs>, <Xt>, [<Xn|SP>]
```

64-bit LDCLRAL variant

Applies when size == 11 && A == 1 && R == 1.

```
LDCLRAL <Xs>, <Xt>, [<Xn|SP>]
```

64-bit LDCLRL variant

Applies when size == 11 && A == 0 && R == 1.

```
LDCLRL <Xs>, <Xt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LSE) then UNDEFINED;

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

constant integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
boolean acquire = A == '1' && Rt != '11111';
boolean release = R == '1';
boolean tagchecked = n != 31;
```

Alias conditions

Alias	is preferred when
STCLR, STCLRL	A == '0' && Rt == '11111'

Assembler symbols

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Wt> Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

<Xs> Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

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

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

Operation

```
bits(64) address;
bits(datasize) value;
bits(datasize) data;

AccessDescriptor accdesc = CreateAccDescAtomicOp(MemAtomicOp_BIC, acquire, release, tagchecked);

value = X[s, datasize];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

bits(datasize) comparevalue = bits(datasize) UNKNOWN;    // Irrelevant when not executing CAS
data = MemAtomic(address, comparevalue, value, accdesc);
```

```
if t != 31 then
    X[t, regsize] = ZeroExtend(data, regsize);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

C6.2.169 LDCLRB, LDCLRAB, LDCLRALB, LDCLRLB

Atomic bit clear on byte in memory atomically loads an 8-bit byte from memory, performs a bitwise AND with the complement of the value held in a register on it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, LDCLRAB and LDCLRALB load from memory with acquire semantics.
- LDCLRLB and LDCLRALB store to memory with release semantics.
- LDCLRB has neither acquire nor release semantics.

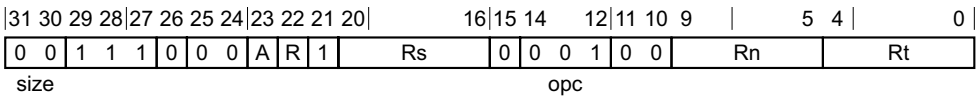
For more information about memory ordering semantics, see *Load-Acquire, Load-AcquirePC, and Store-Release*.

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

This instruction is used by the alias STCLRB, STCLRLB. See *Alias conditions* for details of when each alias is preferred.

Integer

(FEAT_LSE)



LDCLRAB variant

Applies when A == 1 && R == 0.

```
LDCLRAB <Ws>, <Wt>, [<Xn|SP>]
```

LDCLRALB variant

Applies when A == 1 && R == 1.

```
LDCLRALB <Ws>, <Wt>, [<Xn|SP>]
```

LDCLRB variant

Applies when A == 0 && R == 0.

```
LDCLRB <Ws>, <Wt>, [<Xn|SP>]
```

LDCLRLB variant

Applies when A == 0 && R == 1.

```
LDCLRLB <Ws>, <Wt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LSE) then UNDEFINED;

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

boolean acquire = A == '1' && Rt != '11111';
boolean release = R == '1';
boolean tagchecked = n != 31;
```

Alias conditions

Alias	is preferred when
STCLRB, STCLRLB	`A == '0' && Rt == '11111'`

Assembler symbols

`<Ws>` Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

`<Wt>` Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

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

Operation

```
bits(64) address;
bits(8) value;
bits(8) data;

AccessDescriptor accdesc = CreateAccDescAtomicOp(MemAtomicOp_BIC, acquire, release, tagchecked);

value = X[s, 8];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

bits(8) comparevalue = bits(8) UNKNOWN;    // Irrelevant when not executing CAS
data = MemAtomic(address, comparevalue, value, accdesc);

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

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

C6.2.170 LDCLRH, LDCLRAH, LDCLRALH, LDCLRLH

Atomic bit clear on halfword in memory atomically loads a 16-bit halfword from memory, performs a bitwise AND with the complement of the value held in a register on it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, LDCLRAH and LDCLRALH load from memory with acquire semantics.
- LDCLRLH and LDCLRALH store to memory with release semantics.
- LDCLRH has neither acquire nor release semantics.

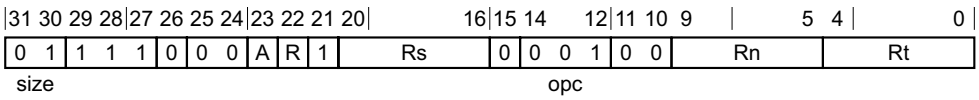
For more information about memory ordering semantics, see *Load-Acquire, Load-AcquirePC, and Store-Release*.

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

This instruction is used by the alias STCLRH, STCLRLH. See *Alias conditions* for details of when each alias is preferred.

Integer

(FEAT_LSE)



LDCLRAH variant

Applies when A == 1 && R == 0.

```
LDCLRAH <Ws>, <Wt>, [<Xn|SP>]
```

LDCLRALH variant

Applies when A == 1 && R == 1.

```
LDCLRALH <Ws>, <Wt>, [<Xn|SP>]
```

LDCLRH variant

Applies when A == 0 && R == 0.

```
LDCLRH <Ws>, <Wt>, [<Xn|SP>]
```

LDCLRLH variant

Applies when A == 0 && R == 1.

```
LDCLRLH <Ws>, <Wt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LSE) then UNDEFINED;

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

boolean acquire = A == '1' && Rt != '11111';
boolean release = R == '1';
boolean tagchecked = n != 31;
```

Alias conditions

Alias	is preferred when
STCLRH, STCLRLH	A == '0' && Rt == '11111'

Assembler symbols

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Wt> Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

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

Operation

```
bits(64) address;
bits(16) value;
bits(16) data;

AccessDescriptor accdesc = CreateAccDescAtomicOp(MemAtomicOp_BIC, acquire, release, tagchecked);

value = X[s, 16];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

bits(16) comparevalue = bits(16) UNKNOWN;    // Irrelevant when not executing CAS
data = MemAtomic(address, comparevalue, value, accdesc);

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

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

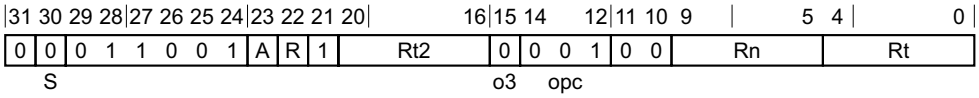
C6.2.171 LDCLRP, LDCLRPA, LDCLRPAL, LDCLRPL

Atomic bit clear on quadword in memory atomically loads a 128-bit quadword from memory, performs a bitwise AND with the complement of the value held in a pair of registers on it, and stores the result back to memory. The value initially loaded from memory is returned in the same pair of registers.

- LDCLRPA and LDCLRPAL load from memory with acquire semantics.
- LDCLRPL and LDCLRPAL store to memory with release semantics.
- LDCLRP has neither acquire nor release semantics.

Integer

(FEAT_LSE128)



LDCLRP variant

Applies when A == 0 && R == 0.

```
LDCLRP <Xt1>, <Xt2>, [<Xn|SP>]
```

LDCLRPA variant

Applies when A == 1 && R == 0.

```
LDCLRPA <Xt1>, <Xt2>, [<Xn|SP>]
```

LDCLRPAL variant

Applies when A == 1 && R == 1.

```
LDCLRPAL <Xt1>, <Xt2>, [<Xn|SP>]
```

LDCLRPL variant

Applies when A == 0 && R == 1.

```
LDCLRPL <Xt1>, <Xt2>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LSE128) then UNDEFINED;
if Rt == '11111' then UNDEFINED;
if Rt2 == '11111' then UNDEFINED;
integer t = UInt(Rt);
integer t2 = UInt(Rt2);
integer n = UInt(Rn);
boolean acquire = A == '1';
boolean release = R == '1';
boolean tagchecked = n != 31;

boolean rt_unknown = FALSE;

if t == t2 then
    Constraint c = ConstrainUnpredictable(Unpredictable_LSE128OVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN rt_unknown = TRUE;    // result is UNKNOWN
```

```
        when Constraint_UNDEF   UNDEFINED;
        when Constraint_NOP     EndOfInstruction();
```

Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*, and particularly *CONSTRAINED UNPREDICTABLE behavior for A64 instructions*.

Assembler symbols

<Xt1>	Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.
<Xt2>	Is the 64-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(64) value1 = X[t, 64];
bits(64) value2 = X[t2, 64];
bits(128) data;
bits(128) store_value;

AccessDescriptor accdesc = CreateAccDescAtomicOp(MemAtomicOp_BIC, acquire, release, tagchecked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

store_value = if BigEndian(accdesc.acctype) then value1:value2 else value2:value1;

bits(128) comparevalue = bits(128) UNKNOWN;    // Irrelevant when not executing CAS
data = MemAtomic(address, comparevalue, store_value, accdesc);

if rt_unknown then
    data = bits(128) UNKNOWN;

if BigEndian(accdesc.acctype) then
    X[t, 64] = data<127:64>;
    X[t2, 64] = data<63:0>;
else
    X[t, 64] = data<63:0>;
    X[t2, 64] = data<127:64>;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

C6.2.172 LDEOR, LDEORA, LDEORAL, LDEORL

Atomic Exclusive-OR on word or doubleword in memory atomically loads a 32-bit word or 64-bit doubleword from memory, performs an exclusive-OR with the value held in a register on it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not one of `WZR` or `XZR`, `LDEORA` and `LDEORAL` load from memory with acquire semantics.
- `LDEORL` and `LDEORAL` store to memory with release semantics.
- `LDEOR` has neither acquire nor release semantics.

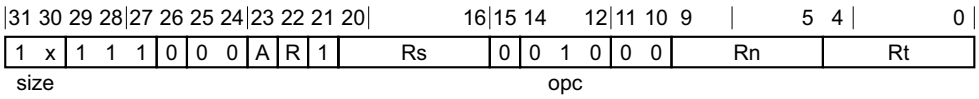
For more information about memory ordering semantics, see *Load-Acquire, Load-AcquirePC, and Store-Release*.

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

This instruction is used by the alias STEOR, STEORL. See *Alias conditions* for details of when each alias is preferred.

Integer

(FEAT_LSE)



32-bit LDEOR variant

Applies when `size == 10 && A == 0 && R == 0`.

```
LDEOR <Ws>, <Wt>, [<Xn|SP>]
```

32-bit LDEORA variant

Applies when `size == 10 && A == 1 && R == 0`.

```
LDEORA <Ws>, <Wt>, [<Xn|SP>]
```

32-bit LDEORAL variant

Applies when `size == 10 && A == 1 && R == 1`.

```
LDEORAL <Ws>, <Wt>, [<Xn|SP>]
```

32-bit LDEORL variant

Applies when `size == 10 && A == 0 && R == 1`.

```
LDEORL <Ws>, <Wt>, [<Xn|SP>]
```

64-bit LDEOR variant

Applies when `size == 11 && A == 0 && R == 0`.

```
LDEOR <Xs>, <Xt>, [<Xn|SP>]
```

64-bit LDEORA variant

Applies when `size == 11 && A == 1 && R == 0`.

```
LDEORA <Xs>, <Xt>, [<Xn|SP>]
```

64-bit LDEORAL variant

Applies when size == 11 && A == 1 && R == 1.

```
LDEORAL <Xs>, <Xt>, [<Xn|SP>]
```

64-bit LDEORL variant

Applies when size == 11 && A == 0 && R == 1.

```
LDEORL <Xs>, <Xt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LSE) then UNDEFINED;

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

constant integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
boolean acquire = A == '1' && Rt != '11111';
boolean release = R == '1';
boolean tagchecked = n != 31;
```

Alias conditions

Alias	is preferred when
STEOR, STEORL	A == '0' && Rt == '11111'

Assembler symbols

<Ws>	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Wt>	Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xs>	Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Xt>	Is the 64-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(datasize) value;
bits(datasize) data;

AccessDescriptor accdesc = CreateAccDescAtomicOp(MemAtomicOp_EOR, acquire, release, tagchecked);

value = X[s, datasize];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

bits(datasize) comparevalue = bits(datasize) UNKNOWN;    // Irrelevant when not executing CAS
data = MemAtomic(address, comparevalue, value, accdesc);
```

```
if t != 31 then
    X[t, regsize] = ZeroExtend(data, regsize);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

C6.2.173 LDEORB, LDEORAB, LDEORALB, LDEORLB

Atomic Exclusive-OR on byte in memory atomically loads an 8-bit byte from memory, performs an exclusive-OR with the value held in a register on it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, LDEORAB and LDEORALB load from memory with acquire semantics.
- LDEORLB and LDEORALB store to memory with release semantics.
- LDEORB has neither acquire nor release semantics.

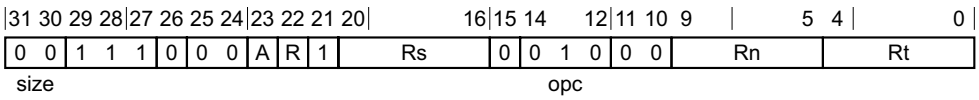
For more information about memory ordering semantics, see *Load-Acquire, Load-AcquirePC, and Store-Release*.

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

This instruction is used by the alias STEORB, STEORLB. See *Alias conditions* for details of when each alias is preferred.

Integer

(FEAT_LSE)



LDEORAB variant

Applies when A == 1 && R == 0.

```
LDEORAB <Ws>, <Wt>, [<Xn|SP>]
```

LDEORALB variant

Applies when A == 1 && R == 1.

```
LDEORALB <Ws>, <Wt>, [<Xn|SP>]
```

LDEORB variant

Applies when A == 0 && R == 0.

```
LDEORB <Ws>, <Wt>, [<Xn|SP>]
```

LDEORLB variant

Applies when A == 0 && R == 1.

```
LDEORLB <Ws>, <Wt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LSE) then UNDEFINED;

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

boolean acquire = A == '1' && Rt != '11111';
boolean release = R == '1';
boolean tagchecked = n != 31;
```

Alias conditions

Alias	is preferred when
STEORB, STEORLB	`A == '0' && Rt == '11111'`

Assembler symbols

`<Ws>` Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

`<Wt>` Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

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

Operation

```
bits(64) address;
bits(8) value;
bits(8) data;

AccessDescriptor accdesc = CreateAccDescAtomicOp(MemAtomicOp_EOR, acquire, release, tagchecked);

value = X[s, 8];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

bits(8) comparevalue = bits(8) UNKNOWN;    // Irrelevant when not executing CAS
data = MemAtomic(address, comparevalue, value, accdesc);

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

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

C6.2.174 LDEORH, LDEORAH, LDEORALH, LDEORLH

Atomic Exclusive-OR on halfword in memory atomically loads a 16-bit halfword from memory, performs an exclusive-OR with the value held in a register on it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, LDEORAH and LDEORALH load from memory with acquire semantics.
- LDEORLH and LDEORALH store to memory with release semantics.
- LDEORH has neither acquire nor release semantics.

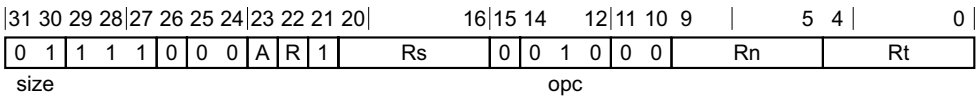
For more information about memory ordering semantics, see *Load-Acquire, Load-AcquirePC, and Store-Release*.

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

This instruction is used by the alias STEORH, STEORLH. See *Alias conditions* for details of when each alias is preferred.

Integer

(FEAT_LSE)



LDEORAH variant

Applies when A == 1 && R == 0.

```
LDEORAH <Ws>, <Wt>, [<Xn|SP>]
```

LDEORALH variant

Applies when A == 1 && R == 1.

```
LDEORALH <Ws>, <Wt>, [<Xn|SP>]
```

LDEORH variant

Applies when A == 0 && R == 0.

```
LDEORH <Ws>, <Wt>, [<Xn|SP>]
```

LDEORLH variant

Applies when A == 0 && R == 1.

```
LDEORLH <Ws>, <Wt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LSE) then UNDEFINED;

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

boolean acquire = A == '1' && Rt != '11111';
boolean release = R == '1';
boolean tagchecked = n != 31;
```

Alias conditions

Alias	is preferred when
STEORH, STEORLH	`A == '0' && Rt == '11111'`

Assembler symbols

`<Ws>` Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

`<Wt>` Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

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

Operation

```
bits(64) address;
bits(16) value;
bits(16) data;

AccessDescriptor accdesc = CreateAccDescAtomicOp(MemAtomicOp_EOR, acquire, release, tagchecked);

value = X[s, 16];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

bits(16) comparevalue = bits(16) UNKNOWN;    // Irrelevant when not executing CAS
data = MemAtomic(address, comparevalue, value, accdesc);

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

Operational information

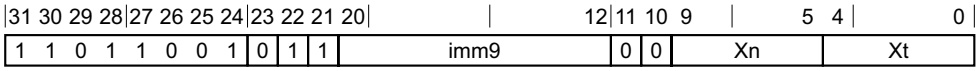
If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

C6.2.175 LDG

Load Allocation Tag loads an Allocation Tag from a memory address, generates a Logical Address Tag from the Allocation Tag and merges it into the destination register. The address used for the load is calculated from the base register and an immediate signed offset scaled by the Tag granule.

Integer

(FEAT_MTE)



Encoding

```
LDG <Xt>, [<Xn|SP>{, #<simm>}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_MTE) then UNDEFINED;
integer t = UInt(Xt);
integer n = UInt(Xn);
bits(64) offset = LSL(SignExtend(imm9, 64), LOG2_TAG_GRANULE);
```

Assembler symbols

<Xt>	Is the 64-bit name of the general-purpose destination register, encoded in the "Xt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Xn" field.
<simm>	Is the optional signed immediate offset, a multiple of 16 in the range -4096 to 4080, defaulting to 0 and encoded in the "imm9" field.

Operation

```
bits(64) address;
bits(4) tag;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

AccessDescriptor accdesc = CreateAccDescLDGSTG(MemOp_LOAD);

address = GenerateAddress(address, offset, accdesc);
address = Align(address, TAG_GRANULE);

tag = AArch64.MemTag[address, accdesc];
X[t, 64] = AArch64.AddressWithAllocationTag(X[t, 64], tag);
```

C6.2.176 LDGM

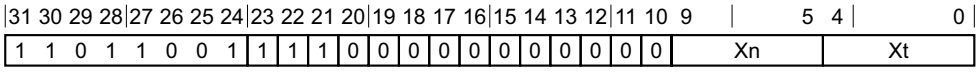
Load Tag Multiple reads a naturally aligned block of N Allocation Tags, where the size of N is identified in GMID_EL1.BS, and writes the Allocation Tag read from address A to the destination register at 4*A<7:4>+3:4*A<7:4>. Bits of the destination register not written with an Allocation Tag are set to 0.

This instruction is UNDEFINED at EL0.

This instruction generates an Unchecked access.

Integer

(FEAT_MTE2)



Encoding

```
LDGM <Xt>, [<Xn|SP>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_MTE2) then UNDEFINED;
integer t = UInt(Xt);
integer n = UInt(Xn);
```

Assembler symbols

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

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

Operation

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

bits(64) data = Zeros(64);
bits(64) address;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

integer size = 4 * (2 ^ (UInt(GMID_EL1.BS)));
address = Align(address, size);
constant integer count = size >> LOG2_TAG_GRANULE;
integer index = UInt(address<LOG2_TAG_GRANULE+3:LOG2_TAG_GRANULE>);
AccessDescriptor accdesc = CreateAccDescLDGSTG(MemOp_LOAD);

for i = 0 to count-1
    bits(4) tag = AArch64.MemTag[address, accdesc];
    Elem[data, index, 4] = tag;
    address = GenerateAddress(address, TAG_GRANULE, accdesc);
    index = index + 1;

X[t, 64] = data;
```

C6.2.177 LDIAPP

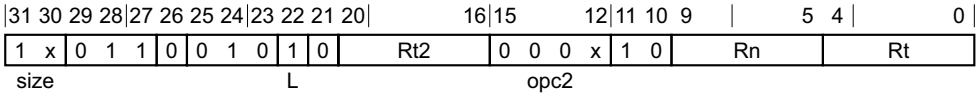
Load-Acquire RCpc ordered Pair of registers calculates an address from a base register value and an optional offset, loads two 32-bit words or two 64-bit doublewords from memory, and writes them to two registers. For information on single-copy atomicity and alignment requirements, see *Requirements for single-copy atomicity* and *Alignment of data accesses*. The instruction also has memory ordering semantics, as described in *Load-Acquire, Load-AcquirePC, and Store-Release*, except that:

- The Memory effects associated with Xt1/Wt1 are Ordered-before the Memory effects associated with Xt2/Wt2.
- There is no ordering requirement, separate from the requirements of a Load-AcquirePC or a Store-Release, created by having a Store-Release followed by a Load-AcquirePC instruction.
- The reading of a value written by a Store-Release by a Load-AcquirePC instruction by the same observer does not make the write of the Store-Release globally observed.

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

Integer

(FEAT_LRCPC3)



32-bit variant

Applies when `size == 10 && opc2 == 0001`.

```
LDIAPP <Wt1>, <Wt2>, [<Xn|SP>]
```

32-bit post-index variant

Applies when `size == 10 && opc2 == 0000`.

```
LDIAPP <Wt1>, <Wt2>, [<Xn|SP>], #8
```

64-bit variant

Applies when `size == 11 && opc2 == 0001`.

```
LDIAPP <Xt1>, <Xt2>, [<Xn|SP>]
```

64-bit post-index variant

Applies when `size == 11 && opc2 == 0000`.

```
LDIAPP <Xt1>, <Xt2>, [<Xn|SP>], #16
```

Decode for all variants of this encoding

```
boolean postindex;
boolean wback;
postindex = opc2<0> == '0';
wback = opc2<0> == '0';
```

Notes for all encodings

LDIAPP has the same CONSTRAINED UNPREDICTABLE behavior as LDP. For information about this CONSTRAINED UNPREDICTABLE behavior, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*, and particularly *LDP and LDIAPP*.

Assembler symbols

<Wt1>	Is the 32-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.
<Wt2>	Is the 32-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.
<Xt1>	Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.
<Xt2>	Is the 64-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Shared decode for all encodings

```
integer offset;
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2);
constant integer scale = 2 + UInt(size<0>);
constant integer datasize = 8 << scale;
offset = if opc2<0> == '0' then (2 << scale) else 0;
boolean tagchecked = wback || n != 31;

boolean rt_unknown = FALSE;
boolean wb_unknown = FALSE;

if wback && (t == n || t2 == n) && n != 31 then
    Constraint c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;    // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;    // writeback is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

if t == t2 then
    Constraint c = ConstrainUnpredictable(Unpredictable_LDPOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN rt_unknown = TRUE;    // result is UNKNOWN
        when Constraint_UNDEF   UNDEFINED;
        when Constraint_NOP     EndOfInstruction();
```

Operation

```
bits(64) address;
bits(64) address2;
bits(datasize) data1;
bits(datasize) data2;
constant integer dbytes = datasize DIV 8;

AccessDescriptor accdesc = CreateAccDescLDAcqPC(tagchecked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
```

```
else
    address = X[n, 64];

if !postindex then
    address = GenerateAddress(address, offset, accdesc);

if IsFeatureImplemented(FEAT_LSE2) then
    bits(2*datasize) full_data;
    accdesc.ispair = TRUE;
    full_data = Mem[address, 2*dbytes, accdesc];
    if BigEndian(accdesc.acctype) then
        data2 = full_data<(datasize-1):0>;
        data1 = full_data<(2*datasize-1):datasize>;
    else
        data1 = full_data<(datasize-1):0>;
        data2 = full_data<(2*datasize-1):datasize>;
else
    address2 = GenerateAddress(address, dbytes, accdesc);
    data1 = Mem[address, dbytes, accdesc];
    data2 = Mem[address2, dbytes, accdesc];
if rt_unknown then
    data1 = bits(datasize) UNKNOWN;
    data2 = bits(datasize) UNKNOWN;

X[t, datasize] = data1;
X[t2, datasize] = data2;

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = GenerateAddress(address, offset, accdesc);
    if n == 31 then
        SP[] = address;
    else
        X[n, 64] = address;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

C6.2.178 LDLAR

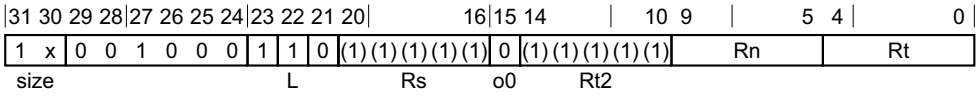
Load LOAcquire Register loads a 32-bit word or 64-bit doubleword from memory, and writes it to a register. The instruction also has memory ordering semantics as described in *LoadLOAcquire, StoreLORelease*. For information about memory accesses, see *Load/store addressing modes*.

Note

For this instruction, if the destination is WZR/XZR, it is impossible for software to observe the presence of the acquire semantic other than its effect on the arrival at endpoints.

No offset

(FEAT_LOR)



32-bit variant

Applies when `size == 10`.

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

64-bit variant

Applies when `size == 11`.

```
LDLAR <Xt>, [<Xn|SP>{,#0}]
```

Decode for all variants of this encoding

```
integer n = UInt(Rn);
integer t = UInt(Rt);

constant integer elsize = 8 << UInt(size);
constant integer regsize = if elsize == 64 then 64 else 32;
boolean tagchecked = n != 31;
```

Assembler symbols

`<Wt>`	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
`<Xt>`	Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
`<Xn|SP>`	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(elsize) data;
constant integer dbytes = elsize DIV 8;

AccessDescriptor accdesc;
accdesc = CreateAccDescLOR(MemOp_LOAD, tagchecked);
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];
```

```
data = Mem[address, dbytes, accdesc];
X[t, regsize] = ZeroExtend(data, regsize);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

C6.2.179 LDLARB

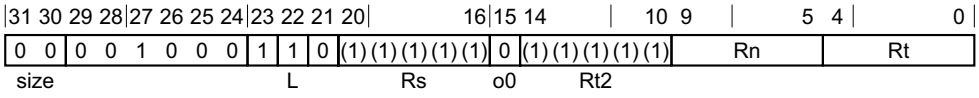
Load LOAcquire Register Byte loads a byte from memory, zero-extends it and writes it to a register. The instruction also has memory ordering semantics as described in *LoadLOAcquire, StoreLORelease*. For information about memory accesses, see *Load/store addressing modes*.

Note

For this instruction, if the destination is WZR/XZR, it is impossible for software to observe the presence of the acquire semantic other than its effect on the arrival at endpoints.

No offset

(FEAT_LOR)



Encoding

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

Decode for this encoding

```
integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tagchecked = n != 31;
```

Assembler symbols

<Wt>	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(8) data;

AccessDescriptor accdesc;
accdesc = CreateAccDescLOR(MemOp_LOAD, tagchecked);
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

data = Mem[address, 1, accdesc];
X[t, 32] = ZeroExtend(data, 32);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

C6.2.180 LDLARH

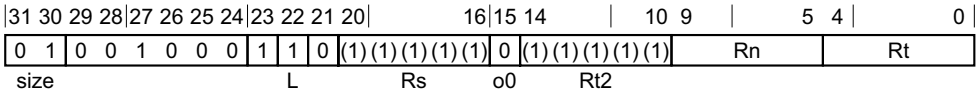
Load LOAcquire Register Halfword loads a halfword from memory, zero-extends it, and writes it to a register. The instruction also has memory ordering semantics as described in *LoadLOAcquire, StoreLORelease*. For information about memory accesses, see *Load/store addressing modes*.

Note

For this instruction, if the destination is WZR/XZR, it is impossible for software to observe the presence of the acquire semantic other than its effect on the arrival at endpoints.

No offset

(FEAT_LOR)



Encoding

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

Decode for this encoding

```
integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tagchecked = n != 31;
```

Assembler symbols

<Wt>	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(16) data;

AccessDescriptor accdesc;
accdesc = CreateAccDescLOR(MemOp_LOAD, tagchecked);
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

data = Mem[address, 2, accdesc];
X[t, 32] = ZeroExtend(data, 32);
```

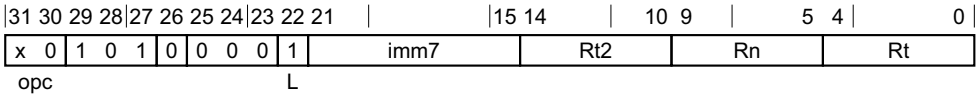
Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

C6.2.181 LDNP

Load Pair of Registers, with non-temporal hint, calculates an address from a base register value and an immediate offset, loads two 32-bit words or two 64-bit doublewords from memory, and writes them to two registers.

For information about memory accesses, see *Load/store addressing modes*. For information about Non-temporal pair instructions, see *Load/store non-temporal pair*.



32-bit variant

Applies when `opc == 00`.

```
LDNP <Wt1>, <Wt2>, [<Xn|SP>{, #<imm>}]
```

64-bit variant

Applies when `opc == 10`.

```
LDNP <Xt1>, <Xt2>, [<Xn|SP>{, #<imm>}]
```

Decode for all variants of this encoding

```
// Empty.
```

Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*, and particularly *LDNP*.

Assembler symbols

`<Wt1>` Is the 32-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.

`<Wt2>` Is the 32-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.

`<Xt1>` Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.

`<Xt2>` Is the 64-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.

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

`<imm>` For the 32-bit variant: is the optional signed immediate byte offset, a multiple of 4 in the range -256 to 252, defaulting to 0 and encoded in the "imm7" field as <imm>/4.

For the 64-bit variant: is the optional signed immediate byte offset, a multiple of 8 in the range -512 to 504, defaulting to 0 and encoded in the "imm7" field as <imm>/8.

Shared decode for all encodings

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2);
if opc<0> == '1' then UNDEFINED;
```

```
integer scale = 2 + UInt(opc<1>);
constant integer datasize = 8 << scale;
bits(64) offset = LSL(SignExtend(imm7, 64), scale);
boolean tagchecked = n != 31;

boolean rt_unknown = FALSE;

if t == t2 then
    Constraint c = ConstrainUnpredictable(Unpredictable_LDPOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN rt_unknown = TRUE;    // result is UNKNOWN
        when Constraint_UNDEF   UNDEFINED;
        when Constraint_NOP     EndOfInstruction();
```

Operation

```
bits(64) address;
bits(64) address2;
bits(datasize) data1;
bits(datasize) data2;
constant integer dbytes = datasize DIV 8;
boolean privileged = PSTATE.EL != EL0;

AccessDescriptor accdesc = CreateAccDescGPR(MemOp_LOAD, TRUE, privileged, tagchecked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

address = GenerateAddress(address, offset, accdesc);

if IsFeatureImplemented(FEAT_LSE2) then
    bits(2*datasize) full_data;
    accdesc.ispair = TRUE;
    full_data = Mem[address, 2*dbytes, accdesc];
    if BigEndian(accdesc.acctype) then
        data2 = full_data<(datasize-1):0>;
        data1 = full_data<(2*datasize-1):datasize>;
    else
        data1 = full_data<(datasize-1):0>;
        data2 = full_data<(2*datasize-1):datasize>;
else
    address2 = GenerateAddress(address, dbytes, accdesc);
    data1 = Mem[address, dbytes, accdesc];
    data2 = Mem[address2, dbytes, accdesc];
if rt_unknown then
    data1 = bits(datasize) UNKNOWN;
    data2 = bits(datasize) UNKNOWN;
X[t, datasize] = data1;
X[t2, datasize] = data2;
```

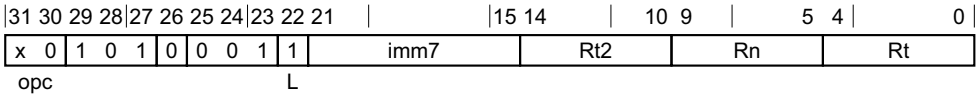
Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

C6.2.182 LDP

Load Pair of Registers calculates an address from a base register value and an immediate offset, loads two 32-bit words or two 64-bit doublewords from memory, and writes them to two registers. For information about memory accesses, see *Load/store addressing modes*.

Post-index



32-bit variant

Applies when `opc == 00`.

```
LDP <Wt1>, <Wt2>, [<Xn|SP>], #<imm>
```

64-bit variant

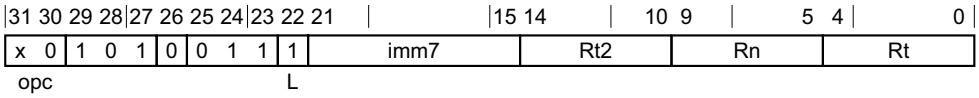
Applies when `opc == 10`.

```
LDP <Xt1>, <Xt2>, [<Xn|SP>], #<imm>
```

Decode for all variants of this encoding

```
boolean wback = TRUE;
boolean postindex = TRUE;
```

Pre-index



32-bit variant

Applies when `opc == 00`.

```
LDP <Wt1>, <Wt2>, [<Xn|SP>, #<imm>]!
```

64-bit variant

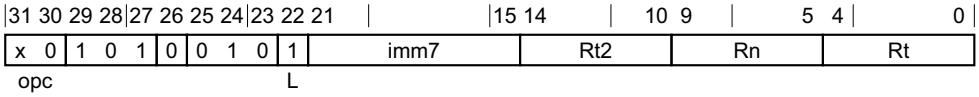
Applies when `opc == 10`.

```
LDP <Xt1>, <Xt2>, [<Xn|SP>, #<imm>]!
```

Decode for all variants of this encoding

```
boolean wback = TRUE;
boolean postindex = FALSE;
```

Signed offset



32-bit variant

Applies when opc == 00.

```
LDP <Wt1>, <Wt2>, [<Xn|SP>{, #<imm>}]
```

64-bit variant

Applies when opc == 10.

```
LDP <Xt1>, <Xt2>, [<Xn|SP>{, #<imm>}]
```

Decode for all variants of this encoding

```
boolean wback = FALSE;
boolean postindex = FALSE;
```

Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*, and particularly *LDP and LDIAPP*.

Assembler symbols

<Wt1>	Is the 32-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.
<Wt2>	Is the 32-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.
<Xt1>	Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.
<Xt2>	Is the 64-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm>	For the 32-bit post-index and 32-bit pre-index variant: is the signed immediate byte offset, a multiple of 4 in the range -256 to 252, encoded in the "imm7" field as <imm>/4. For the 32-bit signed offset variant: is the optional signed immediate byte offset, a multiple of 4 in the range -256 to 252, defaulting to 0 and encoded in the "imm7" field as <imm>/4. For the 64-bit post-index and 64-bit pre-index variant: is the signed immediate byte offset, a multiple of 8 in the range -512 to 504, encoded in the "imm7" field as <imm>/8. For the 64-bit signed offset variant: is the optional signed immediate byte offset, a multiple of 8 in the range -512 to 504, defaulting to 0 and encoded in the "imm7" field as <imm>/8.

Shared decode for all encodings

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2);
if L:opc<0> == '01' || opc == '11' then UNDEFINED;
boolean signed = (opc<0> != '0');
integer scale = 2 + UInt(opc<1>);
constant integer datasize = 8 << scale;
bits(64) offset = LSL(SignExtend(imm7, 64), scale);
boolean tagchecked = wback || n != 31;

boolean rt_unknown = FALSE;
boolean wb_unknown = FALSE;

if wback && (t == n || t2 == n) && n != 31 then
    Constraint c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
```

```
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;    // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;    // writeback is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

if t == t2 then
    Constraint c = ConstrainUnpredictable(Unpredictable_LDPOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN rt_unknown = TRUE;    // result is UNKNOWN
        when Constraint_UNDEF   UNDEFINED;
        when Constraint_NOP     EndOfInstruction();
```

Operation for all encodings

```
bits(64) address;
bits(64) address2;
bits(datasize) data1;
bits(datasize) data2;
constant integer dbytes = datasize DIV 8;
boolean privileged = PSTATE.EL != EL0;

AccessDescriptor accdesc = CreateAccDescGPR(MemOp_LOAD, FALSE, privileged, tagchecked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

if !postindex then
    address = GenerateAddress(address, offset, accdesc);

if !signed && IsFeatureImplemented(FEAT_LSE2) then
    bits(2*datasize) full_data;
    accdesc.ispair = TRUE;
    full_data = Mem[address, 2*dbytes, accdesc];
    if BigEndian(accdesc.acctype) then
        data2 = full_data<(datasize-1):0>;
        data1 = full_data<(2*datasize-1):datasize>;
    else
        data1 = full_data<(datasize-1):0>;
        data2 = full_data<(2*datasize-1):datasize>;
else
    address2 = GenerateAddress(address, dbytes, accdesc);
    data1 = Mem[address, dbytes, accdesc];
    data2 = Mem[address2, dbytes, accdesc];
if rt_unknown then
    data1 = bits(datasize) UNKNOWN;
    data2 = bits(datasize) UNKNOWN;
if signed then
    X[t, 64] = SignExtend(data1, 64);
    X[t2, 64] = SignExtend(data2, 64);
else
    X[t, datasize] = data1;
    X[t2, datasize] = data2;

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = GenerateAddress(address, offset, accdesc);
    if n == 31 then
        SP[] = address;
```

```
else
    X[n, 64] = address;
```

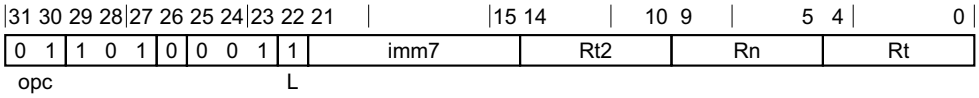
Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

C6.2.183 LDPSW

Load Pair of Registers Signed Word calculates an address from a base register value and an immediate offset, loads two 32-bit words from memory, sign-extends them, and writes them to two registers. For information about memory accesses, see *Load/store addressing modes*.

Post-index



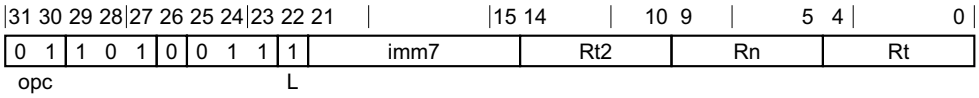
Encoding

```
LDPSW <Xt1>, <Xt2>, [<Xn|SP>], #<imm>
```

Decode for this encoding

```
boolean wback = TRUE;
boolean postindex = TRUE;
```

Pre-index



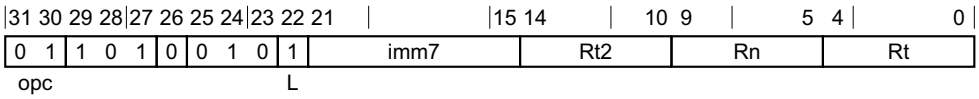
Encoding

```
LDPSW <Xt1>, <Xt2>, [<Xn|SP>, #<imm>]!
```

Decode for this encoding

```
boolean wback = TRUE;
boolean postindex = FALSE;
```

Signed offset



Encoding

```
LDPSW <Xt1>, <Xt2>, [<Xn|SP>{, #<imm>}]
```

Decode for this encoding

```
boolean wback = FALSE;
boolean postindex = FALSE;
```

Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*, and particularly *LDPSW*.

Assembler symbols

<Xt1>	Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.
<Xt2>	Is the 64-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm>	For the post-index and pre-index variant: is the signed immediate byte offset, a multiple of 4 in the range -256 to 252, encoded in the "imm7" field as <imm>/4. For the signed offset variant: is the optional signed immediate byte offset, a multiple of 4 in the range -256 to 252, defaulting to 0 and encoded in the "imm7" field as <imm>/4.

Shared decode for all encodings

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2);
bits(64) offset = LSL(SignExtend(imm7, 64), 2);
boolean tagchecked = wback || n != 31;

boolean rt_unknown = FALSE;
boolean wb_unknown = FALSE;

if wback && (t == n || t2 == n) && n != 31 then
    Constraint c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;    // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;    // writeback is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

if t == t2 then
    Constraint c = ConstrainUnpredictable(Unpredictable_LDPOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN rt_unknown = TRUE;    // result is UNKNOWN
        when Constraint_UNDEF   UNDEFINED;
        when Constraint_NOP     EndOfInstruction();
```

Operation for all encodings

```
bits(64) address;
bits(64) address2;
bits(32) data1;
bits(32) data2;
boolean privileged = PSTATE.EL != EL0;

AccessDescriptor accdesc = CreateAccDescGPR(MemOp_LOAD, FALSE, privileged, tagchecked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];
```

```
if !postindex then
    address = GenerateAddress(address, offset, accdesc);

address2 = GenerateAddress(address, 4, accdesc);
data1 = Mem[address, 4, accdesc];
data2 = Mem[address2, 4, accdesc];
if rt_unknown then
    data1 = bits(32) UNKNOWN;
    data2 = bits(32) UNKNOWN;
X[t, 64] = SignExtend(data1, 64);
X[t2, 64] = SignExtend(data2, 64);
if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = GenerateAddress(address, offset, accdesc);
    if n == 31 then
        SP[] = address;
    else
        X[n, 64] = address;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

C6.2.184 LDR (immediate)

Load Register (immediate) loads a word or doubleword from memory and writes it to a register. The address that is used for the load is calculated from a base register and an immediate offset. For information about memory accesses, see *Load/store addressing modes*. The Unsigned offset variant scales the immediate offset value by the size of the value accessed before adding it to the base register value.

Post-index

32-bit variant

Applies when size == 10.

```
LDR <Wt>, [<Xn|SP>], #<simm>
```

64-bit variant

Applies when size == 11.

```
LDR <Xt>, [<Xn|SP>], #<simm>
```

Decode for all variants of this encoding

```
boolean wback = TRUE;
boolean postindex = TRUE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

Pre-index

32-bit variant

Applies when size == 10.

```
LDR <Wt>, [<Xn|SP>, #<simm>]!
```

64-bit variant

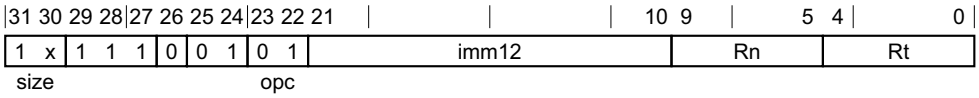
Applies when size == 11.

```
LDR <Xt>, [<Xn|SP>, #<simm>]!
```

Decode for all variants of this encoding

```
boolean wback = TRUE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

Unsigned offset



32-bit variant

Applies when `size == 10`.

```
LDR <Wt>, [<Xn|SP>{, #<pimm>}]
```

64-bit variant

Applies when `size == 11`.

```
LDR <Xt>, [<Xn|SP>{, #<pimm>}]
```

Decode for all variants of this encoding

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = LSL(ZeroExtend(imm12, 64), scale);
```

Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*, and particularly *LDR (immediate)*.

Assembler symbols

`<Wt>`	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
`<Xt>`	Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
`<Xn|SP>`	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
`<simm>`	Is the signed immediate byte offset, in the range -256 to 255, encoded in the "imm9" field.
`<pimm>`	For the 32-bit variant: is the optional positive immediate byte offset, a multiple of 4 in the range 0 to 16380, defaulting to 0 and encoded in the "imm12" field as <pimm>/4. For the 64-bit variant: is the optional positive immediate byte offset, a multiple of 8 in the range 0 to 32760, defaulting to 0 and encoded in the "imm12" field as <pimm>/8.

Shared decode for all encodings

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer regsize;

regsize = if size == '11' then 64 else 32;
constant integer datasize = 8 << scale;
boolean tagchecked = wback || n != 31;

boolean wb_unknown = FALSE;
Constraint c;

if wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
```

```
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;    // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;    // writeback is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();
```

Operation for all encodings

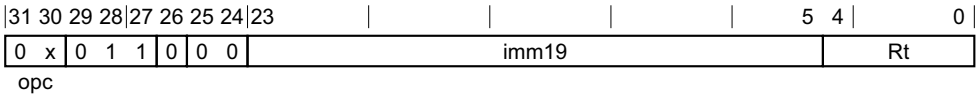
```
bits(64) address;
bits(datasize) data;

boolean privileged = PSTATE.EL != EL0;
AccessDescriptor accdesc = CreateAccDescGPR(MemOp_LOAD, FALSE, privileged, tagchecked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

if !postindex then
    address = GenerateAddress(address, offset, accdesc);

data = Mem[address, datasize DIV 8, accdesc];
X[t, regsize] = ZeroExtend(data, regsize);

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = GenerateAddress(address, offset, accdesc);
    if n == 31 then
        SP[] = address;
    else
        X[n, 64] = address;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

C6.2.185 LDR (literal)

Load Register (literal) calculates an address from the PC value and an immediate offset, loads a word from memory, and writes it to a register. For information about memory accesses, see *Load/store addressing modes*.



32-bit variant

Applies when opc == 00.

```
LDR <Wt>, <label>
```

64-bit variant

Applies when opc == 01.

```
LDR <Xt>, <label>
```

Decode for all variants of this encoding

```
integer t = UInt(Rt);
MemOp memop = if opc == '11' then MemOp_PREFETCH else MemOp_LOAD;
constant integer size = 4 << UInt(opc<0>);
boolean signed = opc == '10';

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

Assembler symbols

<Wt>	Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xt>	Is the 64-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<label>	Is the program label from which the data is to be loaded. Its offset from the address of this instruction, in the range +/-1MB, is encoded as "imm19" times 4.

Operation

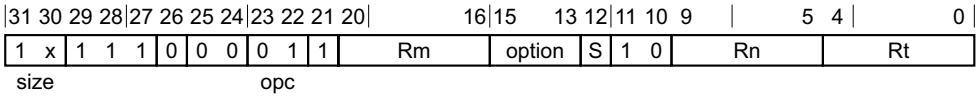
```
bits(64) address = PC64 + offset;
bits(size*8) data;
boolean privileged = PSTATE.EL != EL0;

AccessDescriptor accdesc = CreateAccDescGPR(memop, FALSE, privileged, FALSE);
case memop of
    when MemOp_LOAD
        data = Mem[address, size, accdesc];
        if signed then
            X[t, 64] = SignExtend(data, 64);
        else
            X[t, size*8] = data;

    when MemOp_PREFETCH
        Prefetch(address, t<4:0>);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

C6.2.186 LDR (register)

Load Register (register) calculates an address from a base register value and an offset register value, loads a word from memory, and writes it to a register. The offset register value can optionally be shifted and extended. For information about memory accesses, see *Load/store addressing modes*.



32-bit variant

Applies when `size == 10`.

```
LDR <Wt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

64-bit variant

Applies when `size == 11`.

```
LDR <Xt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

Decode for all variants of this encoding

```
integer scale = UInt(size);
if option<1> == '0' then UNDEFINED;    // sub-word index
ExtendType extend_type = DecodeRegExtend(option);
integer shift = if S == '1' then scale else 0;
```

Assembler symbols

`<Wt>` Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

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

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

`<Wm>` When `option<0>` is set to `0`, is the 32-bit name of the general-purpose index register, encoded in the "Rm" field.

`<Xm>` When `option<0>` is set to `1`, is the 64-bit name of the general-purpose index register, encoded in the "Rm" field.

`<extend>` Is the index extend/shift specifier, defaulting to LSL, and which must be omitted for the LSL option when <amount> is omitted. encoded in the "option" field. It can have the following values:

- `UXTW` when `option = 010`
- `LSL` when `option = 011`
- `SXTW` when `option = 110`
- `SXTX` when `option = 111`

`<amount>` For the 32-bit variant: is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in the "S" field. It can have the following values:

- `#0` when `S = 0`
- `#2` when `S = 1`

For the 64-bit variant: is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in the "S" field. It can have the following values:

`#0` when `S = 0`

`#3` when `S = 1`

Shared decode for all encodings

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer m = UInt(Rm);
integer regsize;

regsize = if size == '11' then 64 else 32;
constant integer datasize = 8 << scale;
```

Operation

```
bits(64) offset = ExtendReg(m, extend_type, shift, 64);
bits(64) address;
bits(datasize) data;

boolean privileged = PSTATE.EL != EL0;
AccessDescriptor accdesc = CreateAccDescGPR(MemOp_LOAD, FALSE, privileged, TRUE);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

address = GenerateAddress(address, offset, accdesc);

data = Mem[address, datasize DIV 8, accdesc];
X[t, regsize] = ZeroExtend(data, regsize);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

C6.2.187 LDRAA, LDRAB

Load Register, with pointer authentication. This instruction authenticates an address from a base register using a modifier of zero and the specified key, adds an immediate offset to the authenticated address, and loads a 64-bit doubleword from memory at this resulting address into a register.

Key A is used for `LDRAA`. Key B is used for `LDRAB`.

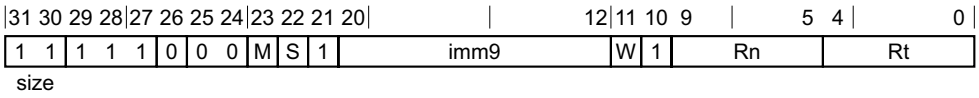
If the authentication passes, the PE behaves the same as for an `LDR` instruction. For information on behavior if the authentication fails, see *Faulting on pointer authentication*.

The authenticated address is not written back to the base register, unless the pre-indexed variant of the instruction is used. In this case, the address that is written back to the base register does not include the pointer authentication code.

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

Unscaled offset

(FEAT_PAuth)



Key A, offset variant

Applies when `M == 0 && W == 0`.

```
LDRAA <Xt>, [<Xn|SP>{, #<simm>}]
```

Key A, pre-indexed variant

Applies when `M == 0 && W == 1`.

```
LDRAA <Xt>, [<Xn|SP>{, #<simm>}]!
```

Key B, offset variant

Applies when `M == 1 && W == 0`.

```
LDRAB <Xt>, [<Xn|SP>{, #<simm>}]
```

Key B, pre-indexed variant

Applies when `M == 1 && W == 1`.

```
LDRAB <Xt>, [<Xn|SP>{, #<simm>}]!
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_PAuth) then UNDEFINED;
integer t = UInt(Rt);
integer n = UInt(Rn);
boolean wback = (W == '1');
boolean use_key_a = (M == '0');
bits(10) S10 = S:imm9;
bits(64) offset = LSL(SignExtend(S10, 64), 3);
boolean tagchecked = wback || n != 31;
```

Assembler symbols

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

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

<simm> Is the optional signed immediate byte offset, a multiple of 8 in the range -4096 to 4088, defaulting to 0 and encoded in the "S:imm9" field as <simm>/8.

Operation

```
bits(64) address;
bits(64) data;
boolean privileged = PSTATE.EL != EL0;
boolean wb_unknown = FALSE;

AccessDescriptor accdesc = CreateAccDescGPR(MemOp_LOAD, FALSE, privileged, tagchecked);
if wback && n == t && n != 31 then
    Constraint c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;    // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;    // writeback is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

if n == 31 then
    address = SP[];
else
    address = X[n, 64];

if use_key_a then
    address = AuthDA(address, X[31, 64], TRUE);
else
    address = AuthDB(address, X[31, 64], TRUE);

if n == 31 then
    CheckSPAlignment();

address = GenerateAddress(address, offset, accdesc);
data = Mem[address, 8, accdesc];
X[t, 64] = data;

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    if n == 31 then
        SP[] = address;
    else
        X[n, 64] = address;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

C6.2.188 LDRB (immediate)

Load Register Byte (immediate) loads a byte from memory, zero-extends it, and writes the result to a register. The address that is used for the load is calculated from a base register and an immediate offset. For information about memory accesses, see *Load/store addressing modes*.

Post-index

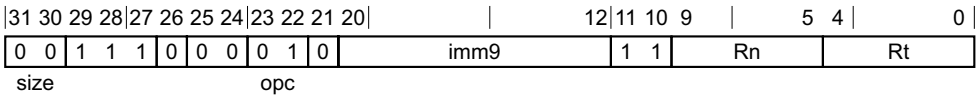
Encoding

```
LDRB <Wt>, [<Xn|SP>], #<simm>
```

Decode for this encoding

```
boolean wback = TRUE;
boolean postindex = TRUE;
bits(64) offset = SignExtend(imm9, 64);
```

Pre-index



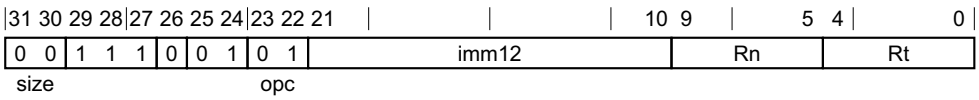
Encoding

```
LDRB <Wt>, [<Xn|SP>, #<simm>]!
```

Decode for this encoding

```
boolean wback = TRUE;
boolean postindex = FALSE;
bits(64) offset = SignExtend(imm9, 64);
```

Unsigned offset



Encoding

```
LDRB <Wt>, [<Xn|SP>{, #<pimm>}]
```

Decode for this encoding

```
boolean wback = FALSE;
boolean postindex = FALSE;
bits(64) offset = LSL(ZeroExtend(imm12, 64), 0);
```

Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*, and particularly *LDRB (immediate)*.

Assembler symbols

<Wt>	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<simm>	Is the signed immediate byte offset, in the range -256 to 255, encoded in the "imm9" field.
<pimm>	Is the optional positive immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the "imm12" field.

Shared decode for all encodings

```
integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tagchecked = wback || n != 31;

boolean wb_unknown = FALSE;
Constraint c;

if wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;    // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;    // writeback is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();
```

Operation for all encodings

```
bits(64) address;
bits(8) data;

boolean privileged = PSTATE.EL != EL0;
AccessDescriptor accdesc = CreateAccDescGPR(MemOp_LOAD, FALSE, privileged, tagchecked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

if !postindex then
    address = GenerateAddress(address, offset, accdesc);

data = Mem[address, 1, accdesc];
X[t, 32] = ZeroExtend(data, 32);

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = GenerateAddress(address, offset, accdesc);
    if n == 31 then
        SP[] = address;
```

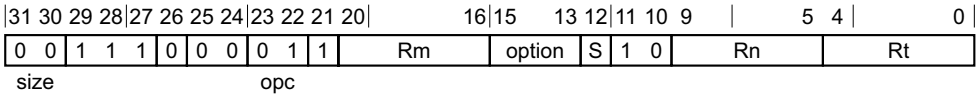
```
else
    X[n, 64] = address;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

C6.2.189 LDRB (register)

Load Register Byte (register) calculates an address from a base register value and an offset register value, loads a byte from memory, zero-extends it, and writes it to a register. For information about memory accesses, see *Load/store addressing modes*.



Extended register variant

Applies when `option != 011`.

```
LDRB <Wt>, [<Xn|SP>, (<Wm>|<Xm>), <extend> {<amount>}]
```

Shifted register variant

Applies when `option == 011`.

```
LDRB <Wt>, [<Xn|SP>, <Xm>{, LSL <amount>}]
```

Decode for all variants of this encoding

```
if option<1> == '0' then UNDEFINED;    // sub-word index
ExtendType extend_type = DecodeRegExtend(option);
```

Assembler symbols

`<Wt>` Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

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

`<Wm>` When `option<0>` is set to `0`, is the 32-bit name of the general-purpose index register, encoded in the "Rm" field.

`<Xm>` When `option<0>` is set to `1`, is the 64-bit name of the general-purpose index register, encoded in the "Rm" field.

`<extend>` Is the index extend specifier, encoded in the "option" field. It can have the following values:

- `UXTW` when `option = 010`
- `SXTW` when `option = 110`
- `SXTX` when `option = 111`

`<amount>` Is the index shift amount, it must be `#0`, encoded in "S" as `0` if omitted, or as `1` if present.

Shared decode for all encodings

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

Operation

```
bits(64) offset = ExtendReg(m, extend_type, 0, 64);
bits(64) address;
bits(8) data;

boolean privileged = PSTATE.EL != EL0;
AccessDescriptor accdesc = CreateAccDescGPR(MemOp_LOAD, FALSE, privileged, TRUE);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

address = GenerateAddress(address, offset, accdesc);

data = Mem[address, 1, accdesc];
X[t, 32] = ZeroExtend(data, 32);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

C6.2.190 LDRH (immediate)

Load Register Halfword (immediate) loads a halfword from memory, zero-extends it, and writes the result to a register. The address that is used for the load is calculated from a base register and an immediate offset. For information about memory accesses, see *Load/store addressing modes*.

Post-index

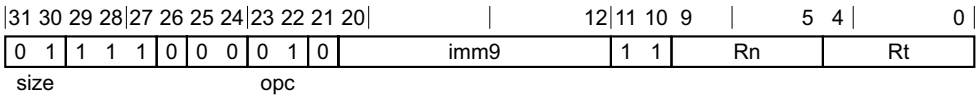
Encoding

```
LDRH <Wt>, [<Xn|SP>], #<simm>
```

Decode for this encoding

```
boolean wback = TRUE;
boolean postindex = TRUE;
bits(64) offset = SignExtend(imm9, 64);
```

Pre-index



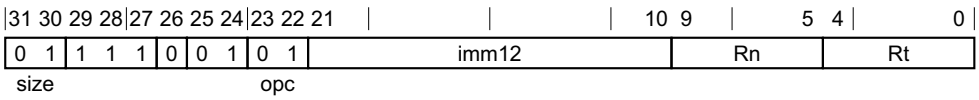
Encoding

```
LDRH <Wt>, [<Xn|SP>, #<simm>]!
```

Decode for this encoding

```
boolean wback = TRUE;
boolean postindex = FALSE;
bits(64) offset = SignExtend(imm9, 64);
```

Unsigned offset



Encoding

```
LDRH <Wt>, [<Xn|SP>{, #<pimm>}]
```

Decode for this encoding

```
boolean wback = FALSE;
boolean postindex = FALSE;
bits(64) offset = LSL(ZeroExtend(imm12, 64), 1);
```

Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*, and particularly *LDRH (immediate)*.

Assembler symbols

<Wt>	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<simm>	Is the signed immediate byte offset, in the range -256 to 255, encoded in the "imm9" field.
<pimm>	Is the optional positive immediate byte offset, a multiple of 2 in the range 0 to 8190, defaulting to 0 and encoded in the "imm12" field as <pimm>/2.

Shared decode for all encodings

```
integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tagchecked = wback || n != 31;

boolean wb_unknown = FALSE;
Constraint c;

if wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;    // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;    // writeback is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();
```

Operation for all encodings

```
bits(64) address;
bits(16) data;

boolean privileged = PSTATE.EL != EL0;
AccessDescriptor accdesc = CreateAccDescGPR(MemOp_LOAD, FALSE, privileged, tagchecked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

if !postindex then
    address = GenerateAddress(address, offset, accdesc);

data = Mem[address, 2, accdesc];
X[t, 32] = ZeroExtend(data, 32);

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = GenerateAddress(address, offset, accdesc);
    if n == 31 then
        SP[] = address;
```

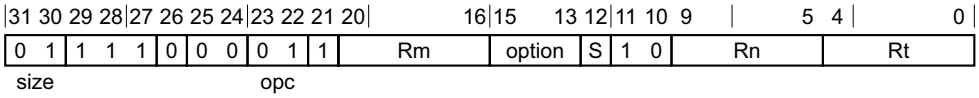
```
else
    X[n, 64] = address;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

C6.2.191 LDRH (register)

Load Register Halfword (register) calculates an address from a base register value and an offset register value, loads a halfword from memory, zero-extends it, and writes it to a register. For information about memory accesses, see *Load/store addressing modes*.



Encoding

```
LDRH <Wt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

Decode for this encoding

```
if option<1> == '0' then UNDEFINED;    // sub-word index
ExtendType extend_type = DecodeRegExtend(option);
integer shift = if S == '1' then 1 else 0;
```

Assembler symbols

`<Wt>` Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

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

`<Wm>` When `option<0>` is set to `0`, is the 32-bit name of the general-purpose index register, encoded in the "Rm" field.

`<Xm>` When `option<0>` is set to `1`, is the 64-bit name of the general-purpose index register, encoded in the "Rm" field.

`<extend>` Is the index extend/shift specifier, defaulting to LSL, and which must be omitted for the LSL option when <amount> is omitted. encoded in the "option" field. It can have the following values:

- `UXTW` when `option = 010`
- `LSL` when `option = 011`
- `SXTW` when `option = 110`
- `SXTX` when `option = 111`

`<amount>` Is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in the "S" field. It can have the following values:

- `#0` when `S = 0`
- `#1` when `S = 1`

Shared decode for all encodings

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

Operation

```
bits(64) offset = ExtendReg(m, extend_type, shift, 64);
bits(64) address;
bits(16) data;

boolean privileged = PSTATE.EL != EL0;
AccessDescriptor accdesc = CreateAccDescGPR(MemOp_LOAD, FALSE, privileged, TRUE);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

address = GenerateAddress(address, offset, accdesc);

data = Mem[address, 2, accdesc];
X[t, 32] = ZeroExtend(data, 32);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

C6.2.192 LDRSB (immediate)

Load Register Signed Byte (immediate) loads a byte from memory, sign-extends it to either 32 bits or 64 bits, and writes the result to a register. The address that is used for the load is calculated from a base register and an immediate offset. For information about memory accesses, see *Load/store addressing modes*.

Post-index

32-bit variant

Applies when `opc == 11`.

```
LDRSB <Wt>, [<Xn|SP>], #<simm>
```

64-bit variant

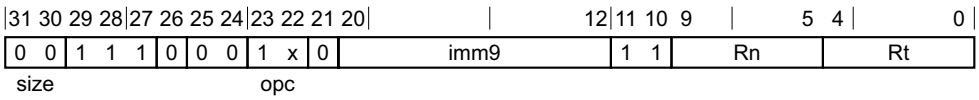
Applies when `opc == 10`.

```
LDRSB <Xt>, [<Xn|SP>], #<simm>
```

Decode for all variants of this encoding

```
boolean wback = TRUE;
boolean postindex = TRUE;
bits(64) offset = SignExtend(imm9, 64);
```

Pre-index



32-bit variant

Applies when `opc == 11`.

```
LDRSB <Wt>, [<Xn|SP>, #<simm>]!
```

64-bit variant

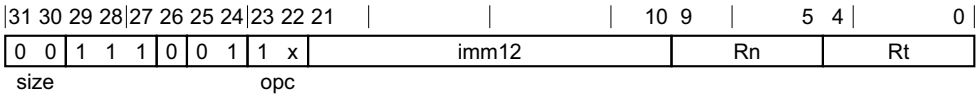
Applies when `opc == 10`.

```
LDRSB <Xt>, [<Xn|SP>, #<simm>]!
```

Decode for all variants of this encoding

```
boolean wback = TRUE;
boolean postindex = FALSE;
bits(64) offset = SignExtend(imm9, 64);
```

Unsigned offset



32-bit variant

Applies when opc == 11.

```
LDRSB <Wt>, [<Xn|SP>{, #<pimm>}]
```

64-bit variant

Applies when opc == 10.

```
LDRSB <Xt>, [<Xn|SP>{, #<pimm>}]
```

Decode for all variants of this encoding

```
boolean wback = FALSE;
boolean postindex = FALSE;
bits(64) offset = LSL(ZeroExtend(imm12, 64), 0);
```

Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*, and particularly *LDRSB (immediate)*.

Assembler symbols

<Wt>	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xt>	Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<simm>	Is the signed immediate byte offset, in the range -256 to 255, encoded in the "imm9" field.
<pimm>	Is the optional positive immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the "imm12" field.

Shared decode for all encodings

```
integer n = UInt(Rn);
integer t = UInt(Rt);
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = 32;
    signed = FALSE;
else
    // sign-extending load
    memop = MemOp_LOAD;
    regsize = if opc<0> == '1' then 32 else 64;
    signed = TRUE;

boolean tagchecked = memop != MemOp_PREFETCH && (wback || n != 31);
```

```
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;
Constraint c;

if memop == MemOp_LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;    // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;    // writeback is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE    rt_unknown = FALSE;    // value stored is original value
        when Constraint_UNKNOWN rt_unknown = TRUE;    // value stored is UNKNOWN
        when Constraint_UNDEF   UNDEFINED;
        when Constraint_NOP     EndOfInstruction();
```

Operation for all encodings

```
bits(64) address;
bits(8) data;

boolean privileged = PSTATE.EL != EL0;
AccessDescriptor accdesc = CreateAccDescGPR(memop, FALSE, privileged, tagchecked);

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

if !postindex then
    address = GenerateAddress(address, offset, accdesc);

case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(8) UNKNOWN;
        else
            data = X[t, 8];
        Mem[address, 1, accdesc] = data;

    when MemOp_LOAD
        data = Mem[address, 1, accdesc];
        if signed then
            X[t, regsize] = SignExtend(data, regsize);
        else
            X[t, regsize] = ZeroExtend(data, regsize);

    when MemOp_PREFETCH
        Prefetch(address, t<4:0>);

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = GenerateAddress(address, offset, accdesc);
    if n == 31 then
        SP[] = address;
```

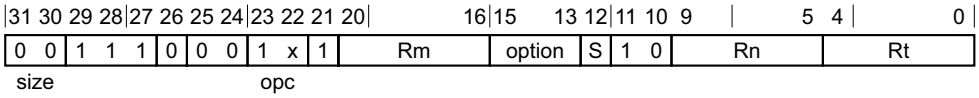
```
else
    X[n, 64] = address;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

C6.2.193 LDRSB (register)

Load Register Signed Byte (register) calculates an address from a base register value and an offset register value, loads a byte from memory, sign-extends it, and writes it to a register. For information about memory accesses, see *Load/store addressing modes*.



32-bit with extended register offset variant

Applies when opc == 11 && option != 011.

```
LDRSB <Wt>, [<Xn|SP>, (<Wm>|<Xm>), <extend> {<amount>}]
```

32-bit with shifted register offset variant

Applies when opc == 11 && option == 011.

```
LDRSB <Wt>, [<Xn|SP>, <Xm>{, LSL <amount>}]
```

64-bit with extended register offset variant

Applies when opc == 10 && option != 011.

```
LDRSB <Xt>, [<Xn|SP>, (<Wm>|<Xm>), <extend> {<amount>}]
```

64-bit with shifted register offset variant

Applies when opc == 10 && option == 011.

```
LDRSB <Xt>, [<Xn|SP>, <Xm>{, LSL <amount>}]
```

Decode for all variants of this encoding

```
if option<1> == '0' then UNDEFINED;    // sub-word index
ExtendType extend_type = DecodeRegExtend(option);
```

Assembler symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

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

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

<Wm> When option<0> is set to 0, is the 32-bit name of the general-purpose index register, encoded in the "Rm" field.

<Xm> When option<0> is set to 1, is the 64-bit name of the general-purpose index register, encoded in the "Rm" field.

<extend> Is the index extend specifier, encoded in the "option" field. It can have the following values:

- UXTW when option = 010
- SXTW when option = 110
- SXTX when option = 111

<amount> Is the index shift amount, it must be #0, encoded in "S" as 0 if omitted, or as 1 if present.

Shared decode for all encodings

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer m = UInt(Rm);
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = 32;
    signed = FALSE;
else
    // sign-extending load
    memop = MemOp_LOAD;
    regsize = if opc<0> == '1' then 32 else 64;
    signed = TRUE;

boolean tagchecked = memop != MemOp_PREFETCH;
```

Operation

```
bits(64) offset = ExtendReg(m, extend_type, 0, 64);
bits(64) address;
bits(8) data;

boolean privileged = PSTATE.EL != EL0;
AccessDescriptor accdesc = CreateAccDescGPR(memop, FALSE, privileged, tagchecked);

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

address = GenerateAddress(address, offset, accdesc);

case memop of
    when MemOp_STORE
        data = X[t, 8];
        Mem[address, 1, accdesc] = data;

    when MemOp_LOAD
        data = Mem[address, 1, accdesc];
        if signed then
            X[t, regsize] = SignExtend(data, regsize);
        else
            X[t, regsize] = ZeroExtend(data, regsize);

    when MemOp_PREFETCH
        Prefetch(address, t<4:0>);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

C6.2.194 LDRSH (immediate)

Load Register Signed Halfword (immediate) loads a halfword from memory, sign-extends it to 32 bits or 64 bits, and writes the result to a register. The address that is used for the load is calculated from a base register and an immediate offset. For information about memory accesses, see *Load/store addressing modes*.

Post-index

32-bit variant

Applies when opc == 11.

```
LDRSH <Wt>, [<Xn|SP>], #<simm>
```

64-bit variant

Applies when opc == 10.

```
LDRSH <Xt>, [<Xn|SP>], #<simm>
```

Decode for all variants of this encoding

```
boolean wback = TRUE;
boolean postindex = TRUE;
bits(64) offset = SignExtend(imm9, 64);
```

Pre-index

32-bit variant

Applies when opc == 11.

```
LDRSH <Wt>, [<Xn|SP>, #<simm>]!
```

64-bit variant

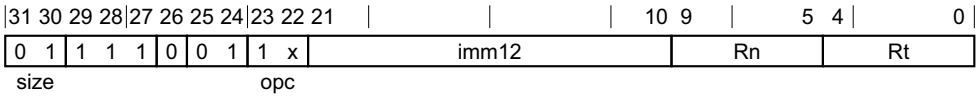
Applies when opc == 10.

```
LDRSH <Xt>, [<Xn|SP>, #<simm>]!
```

Decode for all variants of this encoding

```
boolean wback = TRUE;
boolean postindex = FALSE;
bits(64) offset = SignExtend(imm9, 64);
```

Unsigned offset



32-bit variant

Applies when opc == 11.

```
LDRSH <Wt>, [<Xn|SP>{, #<pimm>}]
```

64-bit variant

Applies when opc == 10.

```
LDRSH <Xt>, [<Xn|SP>{, #<pimm>}]
```

Decode for all variants of this encoding

```
boolean wback = FALSE;
boolean postindex = FALSE;
bits(64) offset = LSL(ZeroExtend(imm12, 64), 1);
```

Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*, and particularly *LDRSH (immediate)*.

Assembler symbols

<Wt>	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xt>	Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<simm>	Is the signed immediate byte offset, in the range -256 to 255, encoded in the "imm9" field.
<pimm>	Is the optional positive immediate byte offset, a multiple of 2 in the range 0 to 8190, defaulting to 0 and encoded in the "imm12" field as <pimm>/2.

Shared decode for all encodings

```
integer n = UInt(Rn);
integer t = UInt(Rt);
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = 32;
    signed = FALSE;
else
    // sign-extending load
    memop = MemOp_LOAD;
    regsize = if opc<0> == '1' then 32 else 64;
    signed = TRUE;

boolean tagchecked = memop != MemOp_PREFETCH && (wback || n != 31);
```

```
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;
Constraint c;

if memop == MemOp_LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;    // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;    // writeback is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE    rt_unknown = FALSE;    // value stored is original value
        when Constraint_UNKNOWN rt_unknown = TRUE;    // value stored is UNKNOWN
        when Constraint_UNDEF   UNDEFINED;
        when Constraint_NOP     EndOfInstruction();
```

Operation for all encodings

```
bits(64) address;
bits(16) data;

boolean privileged = PSTATE.EL != EL0;
AccessDescriptor accdesc = CreateAccDescGPR(memop, FALSE, privileged, tagchecked);

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

if !postindex then
    address = GenerateAddress(address, offset, accdesc);

case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(16) UNKNOWN;
        else
            data = X[t, 16];
        Mem[address, 2, accdesc] = data;

    when MemOp_LOAD
        data = Mem[address, 2, accdesc];
        if signed then
            X[t, regsize] = SignExtend(data, regsize);
        else
            X[t, regsize] = ZeroExtend(data, regsize);

    when MemOp_PREFETCH
        Prefetch(address, t<4:0>);

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = GenerateAddress(address, offset, accdesc);
    if n == 31 then
        SP[] = address;
```

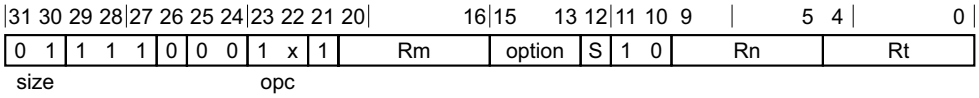
```
else
    X[n, 64] = address;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

C6.2.195 LDRSH (register)

Load Register Signed Halfword (register) calculates an address from a base register value and an offset register value, loads a halfword from memory, sign-extends it, and writes it to a register. For information about memory accesses, see *Load/store addressing modes*.



32-bit variant

Applies when opc == 11.

```
LDRSH <Wt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

64-bit variant

Applies when opc == 10.

```
LDRSH <Xt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

Decode for all variants of this encoding

```
if option<1> == '0' then UNDEFINED;    // sub-word index
ExtendType extend_type = DecodeRegExtend(option);
integer shift = if S == '1' then 1 else 0;
```

Assembler symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

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

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

<Wm> When option<0> is set to 0, is the 32-bit name of the general-purpose index register, encoded in the "Rm" field.

<Xm> When option<0> is set to 1, is the 64-bit name of the general-purpose index register, encoded in the "Rm" field.

<extend> Is the index extend/shift specifier, defaulting to LSL, and which must be omitted for the LSL option when <amount> is omitted. encoded in the "option" field. It can have the following values:

- UXTW when option = 010
- LSL when option = 011
- SXTW when option = 110
- SXTX when option = 111

<amount> Is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in the "S" field. It can have the following values:

- #0 when S = 0
- #1 when S = 1

Shared decode for all encodings

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer m = UInt(Rm);
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = 32;
    signed = FALSE;
else
    // sign-extending load
    memop = MemOp_LOAD;
    regsize = if opc<0> == '1' then 32 else 64;
    signed = TRUE;

boolean tagchecked = memop != MemOp_PREFETCH;
```

Operation

```
bits(64) offset = ExtendReg(m, extend_type, shift, 64);
bits(64) address;
bits(16) data;

boolean privileged = PSTATE.EL != EL0;
AccessDescriptor accdesc = CreateAccDescGPR(memop, FALSE, privileged, tagchecked);

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

address = GenerateAddress(address, offset, accdesc);

case memop of
    when MemOp_STORE
        data = X[t, 16];
        Mem[address, 2, accdesc] = data;

    when MemOp_LOAD
        data = Mem[address, 2, accdesc];
        if signed then
            X[t, regsize] = SignExtend(data, regsize);
        else
            X[t, regsize] = ZeroExtend(data, regsize);

    when MemOp_PREFETCH
        Prefetch(address, t<4:0>);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

C6.2.196 LDRSW (immediate)

Load Register Signed Word (immediate) loads a word from memory, sign-extends it to 64 bits, and writes the result to a register. The address that is used for the load is calculated from a base register and an immediate offset. For information about memory accesses, see *Load/store addressing modes*.

Post-index

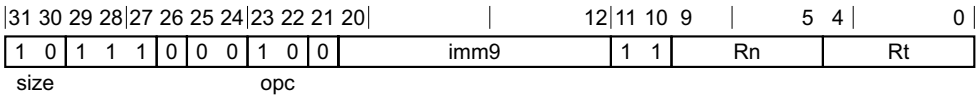
Encoding

```
LDRSW <Xt>, [<Xn|SP>], #<simm>
```

Decode for this encoding

```
boolean wback = TRUE;
boolean postindex = TRUE;
bits(64) offset = SignExtend(imm9, 64);
```

Pre-index



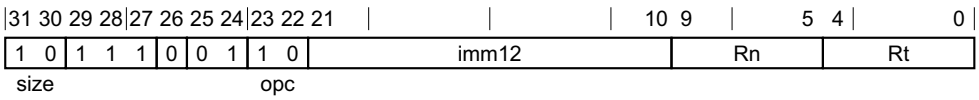
Encoding

```
LDRSW <Xt>, [<Xn|SP>, #<simm>]!
```

Decode for this encoding

```
boolean wback = TRUE;
boolean postindex = FALSE;
bits(64) offset = SignExtend(imm9, 64);
```

Unsigned offset



Encoding

```
LDRSW <Xt>, [<Xn|SP>{, #<pimm>}]
```

Decode for this encoding

```
boolean wback = FALSE;
boolean postindex = FALSE;
bits(64) offset = LSL(ZeroExtend(imm12, 64), 2);
```

Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*, and particularly *LDRSW (immediate)*.

Assembler symbols

<Xt>	Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<simm>	Is the signed immediate byte offset, in the range -256 to 255, encoded in the "imm9" field.
<pimm>	Is the optional positive immediate byte offset, a multiple of 4 in the range 0 to 16380, defaulting to 0 and encoded in the "imm12" field as <pimm>/4.

Shared decode for all encodings

```
integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tagchecked = wback || n != 31;

boolean wb_unknown = FALSE;
Constraint c;

if wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;    // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;    // writeback is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();
```

Operation for all encodings

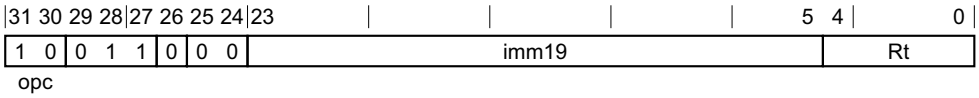
```
bits(64) address;
bits(32) data;

boolean privileged = PSTATE.EL != EL0;
AccessDescriptor accdesc = CreateAccDescGPR(MemOp_LOAD, FALSE, privileged, tagchecked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

if !postindex then
    address = GenerateAddress(address, offset, accdesc);

data = Mem[address, 4, accdesc];
X[t, 64] = SignExtend(data, 64);
if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = GenerateAddress(address, offset, accdesc);
    if n == 31 then
        SP[] = address;
    else
        X[n, 64] = address;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

C6.2.197 LDRSW (literal)

Load Register Signed Word (literal) calculates an address from the PC value and an immediate offset, loads a word from memory, and writes it to a register. For information about memory accesses, see *Load/store addressing modes*.



Encoding

```
LDRSW <Xt>, <label>
```

Decode for this encoding

```
integer t = UInt(Rt);

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

Assembler symbols

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

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

Operation

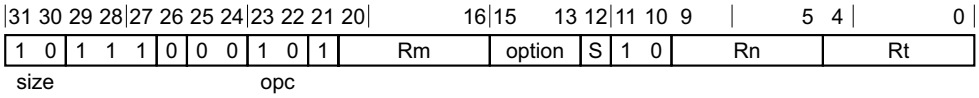
```
bits(64) address = PC64 + offset;
bits(32) data;
boolean privileged = PSTATE.EL != EL0;

AccessDescriptor accdesc = CreateAccDescGPR(MemOp_LOAD, FALSE, privileged, FALSE);
data = Mem[address, 4, accdesc];
X[t, 64] = SignExtend(data, 64);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

C6.2.198 LDRSW (register)

Load Register Signed Word (register) calculates an address from a base register value and an offset register value, loads a word from memory, sign-extends it to form a 64-bit value, and writes it to a register. The offset register value can be shifted left by 0 or 2 bits. For information about memory accesses, see *Load/store addressing modes*.



Encoding

```
LDRSW <Xt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

Decode for this encoding

```
if option<1> == '0' then UNDEFINED;    // sub-word index
ExtendType extend_type = DecodeRegExtend(option);
integer shift = if S == '1' then 2 else 0;
```

Assembler symbols

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

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

<Wm> When option<0> is set to 0, is the 32-bit name of the general-purpose index register, encoded in the "Rm" field.

<Xm> When option<0> is set to 1, is the 64-bit name of the general-purpose index register, encoded in the "Rm" field.

<extend> Is the index extend/shift specifier, defaulting to LSL, and which must be omitted for the LSL option when <amount> is omitted. encoded in the "option" field. It can have the following values:

- UXTW when option = 010
- LSL when option = 011
- SXTW when option = 110
- SXTX when option = 111

<amount> Is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in the "S" field. It can have the following values:

- #0 when S = 0
- #2 when S = 1

Shared decode for all encodings

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

Operation

```
bits(64) offset = ExtendReg(m, extend_type, shift, 64);
bits(64) address;
bits(32) data;

boolean privileged = PSTATE.EL != EL0;
AccessDescriptor accdesc = CreateAccDescGPR(MemOp_LOAD, FALSE, privileged, TRUE);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

address = GenerateAddress(address, offset, accdesc);

data = Mem[address, 4, accdesc];
X[t, 64] = SignExtend(data, 64);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

C6.2.199 LDSET, LDSETA, LDSETAL, LDSETL

Atomic bit set on word or doubleword in memory atomically loads a 32-bit word or 64-bit doubleword from memory, performs a bitwise OR with the value held in a register on it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not one of `WZR` or `XZR`, `LDSETA` and `LDSETAL` load from memory with acquire semantics.
- `LDSETL` and `LDSETAL` store to memory with release semantics.
- `LDSET` has neither acquire nor release semantics.

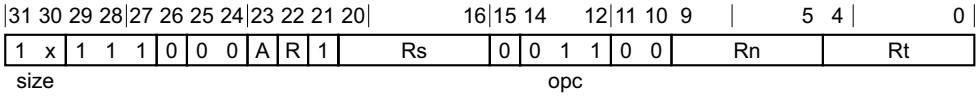
For more information about memory ordering semantics see *Load-Acquire, Load-AcquirePC, and Store-Release*.

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

This instruction is used by the alias STSET, STSETL. See *Alias conditions* for details of when each alias is preferred.

Integer

(FEAT_LSE)



32-bit LDSET variant

Applies when `size == 10 && A == 0 && R == 0`.

```
LDSET <Ws>, <Wt>, [<Xn|SP>]
```

32-bit LDSETA variant

Applies when `size == 10 && A == 1 && R == 0`.

```
LDSETA <Ws>, <Wt>, [<Xn|SP>]
```

32-bit LDSETAL variant

Applies when `size == 10 && A == 1 && R == 1`.

```
LDSETAL <Ws>, <Wt>, [<Xn|SP>]
```

32-bit LDSETL variant

Applies when `size == 10 && A == 0 && R == 1`.

```
LDSETL <Ws>, <Wt>, [<Xn|SP>]
```

64-bit LDSET variant

Applies when `size == 11 && A == 0 && R == 0`.

```
LDSET <Xs>, <Xt>, [<Xn|SP>]
```

64-bit LDSETA variant

Applies when `size == 11 && A == 1 && R == 0`.

```
LDSETA <Xs>, <Xt>, [<Xn|SP>]
```

64-bit LDSETAL variant

Applies when size == 11 && A == 1 && R == 1.

```
LDSETAL <Xs>, <Xt>, [<Xn|SP>]
```

64-bit LDSETL variant

Applies when size == 11 && A == 0 && R == 1.

```
LDSETL <Xs>, <Xt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LSE) then UNDEFINED;

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

constant integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
boolean acquire = A == '1' && Rt != '11111';
boolean release = R == '1';
boolean tagchecked = n != 31;
```

Alias conditions

Alias	is preferred when
STSET, STSETL	A == '0' && Rt == '11111'

Assembler symbols

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Wt> Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

<Xs> Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

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

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

Operation

```
bits(64) address;
bits(datasize) value;
bits(datasize) data;

AccessDescriptor accdesc = CreateAccDescAtomicOp(MemAtomicOp_ORR, acquire, release, tagchecked);

value = X[s, datasize];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

bits(datasize) comparevalue = bits(datasize) UNKNOWN;    // Irrelevant when not executing CAS
data = MemAtomic(address, comparevalue, value, accdesc);
```

```
if t != 31 then
    X[t, regsize] = ZeroExtend(data, regsize);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

C6.2.200 LDSETB, LDSETAB, LDSETALB, LDSETLB

Atomic bit set on byte in memory atomically loads an 8-bit byte from memory, performs a bitwise OR with the value held in a register on it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, LDSETAB and LDSETALB load from memory with acquire semantics.
- LDSETLB and LDSETALB store to memory with release semantics.
- LDSETB has neither acquire nor release semantics.

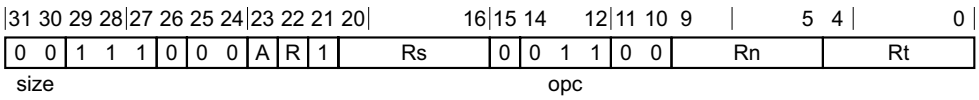
For more information about memory ordering semantics, see *Load-Acquire, Load-AcquirePC, and Store-Release*.

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

This instruction is used by the alias STSETB, STSETLB. See *Alias conditions* for details of when each alias is preferred.

Integer

(FEAT_LSE)



LDSETAB variant

Applies when A == 1 && R == 0.

```
LDSETAB <Ws>, <Wt>, [<Xn|SP>]
```

LDSETALB variant

Applies when A == 1 && R == 1.

```
LDSETALB <Ws>, <Wt>, [<Xn|SP>]
```

LDSETB variant

Applies when A == 0 && R == 0.

```
LDSETB <Ws>, <Wt>, [<Xn|SP>]
```

LDSETLB variant

Applies when A == 0 && R == 1.

```
LDSETLB <Ws>, <Wt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LSE) then UNDEFINED;

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

boolean acquire = A == '1' && Rt != '11111';
boolean release = R == '1';
boolean tagchecked = n != 31;
```

Alias conditions

Alias	is preferred when
STSETB, STSETLB	A == '0' && Rt == '11111'

Assembler symbols

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Wt> Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

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

Operation

```
bits(64) address;
bits(8) value;
bits(8) data;

AccessDescriptor accdesc = CreateAccDescAtomicOp(MemAtomicOp_ORR, acquire, release, tagchecked);

value = X[s, 8];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

bits(8) comparevalue = bits(8) UNKNOWN;    // Irrelevant when not executing CAS
data = MemAtomic(address, comparevalue, value, accdesc);

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

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

C6.2.201 LDSETH, LDSETAH, LDSETALH, LDSETLH

Atomic bit set on halfword in memory atomically loads a 16-bit halfword from memory, performs a bitwise OR with the value held in a register on it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, LDSETAH and LDSETALH load from memory with acquire semantics.
- LDSETLH and LDSETALH store to memory with release semantics.
- LDSETH has neither acquire nor release semantics.

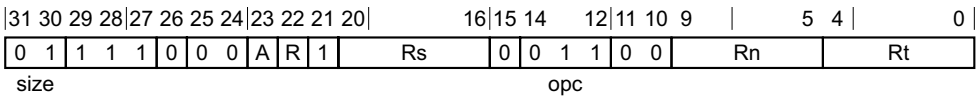
For more information about memory ordering semantics, see *Load-Acquire, Load-AcquirePC, and Store-Release*.

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

This instruction is used by the alias STSETH, STSETLH. See *Alias conditions* for details of when each alias is preferred.

Integer

(FEAT_LSE)



LDSETAH variant

Applies when A == 1 && R == 0.

```
LDSETAH <Ws>, <Wt>, [<Xn|SP>]
```

LDSETALH variant

Applies when A == 1 && R == 1.

```
LDSETALH <Ws>, <Wt>, [<Xn|SP>]
```

LDSETH variant

Applies when A == 0 && R == 0.

```
LDSETH <Ws>, <Wt>, [<Xn|SP>]
```

LDSETLH variant

Applies when A == 0 && R == 1.

```
LDSETLH <Ws>, <Wt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LSE) then UNDEFINED;

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

boolean acquire = A == '1' && Rt != '11111';
boolean release = R == '1';
boolean tagchecked = n != 31;
```

Alias conditions

Alias	is preferred when
STSETH, STSETLH	A == '0' && Rt == '11111'

Assembler symbols

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Wt> Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

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

Operation

```
bits(64) address;
bits(16) value;
bits(16) data;

AccessDescriptor accdesc = CreateAccDescAtomicOp(MemAtomicOp_ORR, acquire, release, tagchecked);

value = X[s, 16];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

bits(16) comparevalue = bits(16) UNKNOWN;    // Irrelevant when not executing CAS
data = MemAtomic(address, comparevalue, value, accdesc);

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

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

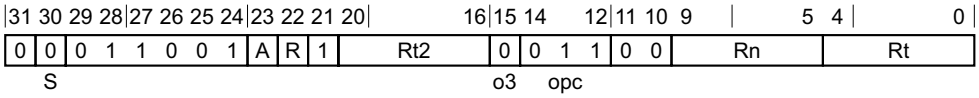
C6.2.202 LDSETP, LDSETPA, LDSETPAL, LDSETPL

Atomic bit set on quadword in memory atomically loads a 128-bit quadword from memory, performs a bitwise OR with the value held in a pair of registers on it, and stores the result back to memory. The value initially loaded from memory is returned in the same pair of registers.

- LDSETPA and LDSETPAL load from memory with acquire semantics.
- LDSETPL and LDSETPAL store to memory with release semantics.
- LDSETP has neither acquire nor release semantics.

Integer

(FEAT_LSE128)



LDSETP variant

Applies when A == 0 && R == 0.

```
LDSETP <Xt1>, <Xt2>, [<Xn|SP>]
```

LDSETPA variant

Applies when A == 1 && R == 0.

```
LDSETPA <Xt1>, <Xt2>, [<Xn|SP>]
```

LDSETPAL variant

Applies when A == 1 && R == 1.

```
LDSETPAL <Xt1>, <Xt2>, [<Xn|SP>]
```

LDSETPL variant

Applies when A == 0 && R == 1.

```
LDSETPL <Xt1>, <Xt2>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LSE128) then UNDEFINED;
if Rt == '11111' then UNDEFINED;
if Rt2 == '11111' then UNDEFINED;
integer t = UInt(Rt);
integer t2 = UInt(Rt2);
integer n = UInt(Rn);

boolean acquire = A == '1';
boolean release = R == '1';
boolean tagchecked = n != 31;

boolean rt_unknown = FALSE;

if t == t2 then
    Constraint c = ConstrainUnpredictable(Unpredictable_LSE128OVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN rt_unknown = TRUE;    // result is UNKNOWN
```

```
        when Constraint_UNDEF   UNDEFINED;
        when Constraint_NOP     EndOfInstruction();
```

Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*, and particularly *CONSTRAINED UNPREDICTABLE behavior for A64 instructions*.

Assembler symbols

<Xt1>	Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.
<Xt2>	Is the 64-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(64) value1 = X[t, 64];
bits(64) value2 = X[t2, 64];
bits(128) data;
bits(128) store_value;

AccessDescriptor accdesc = CreateAccDescAtomicOp(MemAtomicOp_ORR, acquire, release, tagchecked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

store_value = if BigEndian(accdesc.acctype) then value1:value2 else value2:value1;

bits(128) comparevalue = bits(128) UNKNOWN;    // Irrelevant when not executing CAS
data = MemAtomic(address, comparevalue, store_value, accdesc);

if rt_unknown then
    data = bits(128) UNKNOWN;

if BigEndian(accdesc.acctype) then
    X[t, 64] = data<127:64>;
    X[t2, 64] = data<63:0>;
else
    X[t, 64] = data<63:0>;
    X[t2, 64] = data<127:64>;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

C6.2.203 LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL

Atomic signed maximum on word or doubleword in memory atomically loads a 32-bit word or 64-bit doubleword from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as signed numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not one of `WZR` or `XZR`, `LDSMAXA` and `LDSMAXAL` load from memory with acquire semantics.
- `LDSMAXL` and `LDSMAXAL` store to memory with release semantics.
- `LDSMAX` has neither acquire nor release semantics.

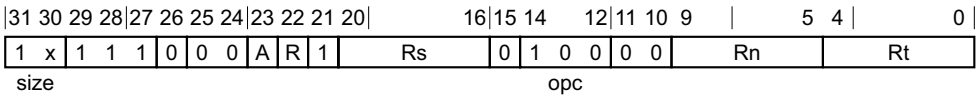
For more information about memory ordering semantics, see *Load-Acquire, Load-AcquirePC, and Store-Release*.

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

This instruction is used by the alias STSMAX, STSMAXL. See *Alias conditions* for details of when each alias is preferred.

Integer

(FEAT_LSE)



32-bit LDSMAX variant

Applies when `size == 10 && A == 0 && R == 0`.

```
LDSMAX <Ws>, <Wt>, [<Xn|SP>]
```

32-bit LDSMAXA variant

Applies when `size == 10 && A == 1 && R == 0`.

```
LDSMAXA <Ws>, <Wt>, [<Xn|SP>]
```

32-bit LDSMAXAL variant

Applies when `size == 10 && A == 1 && R == 1`.

```
LDSMAXAL <Ws>, <Wt>, [<Xn|SP>]
```

32-bit LDSMAXL variant

Applies when `size == 10 && A == 0 && R == 1`.

```
LDSMAXL <Ws>, <Wt>, [<Xn|SP>]
```

64-bit LDSMAX variant

Applies when `size == 11 && A == 0 && R == 0`.

```
LDSMAX <Xs>, <Xt>, [<Xn|SP>]
```

64-bit LDSMAXA variant

Applies when `size == 11 && A == 1 && R == 0`.

```
LDSMAXA <Xs>, <Xt>, [<Xn|SP>]
```

64-bit LDSMAXAL variant

Applies when size == 11 && A == 1 && R == 1.

```
LDSMAXAL <Xs>, <Xt>, [<Xn|SP>]
```

64-bit LDSMAXL variant

Applies when size == 11 && A == 0 && R == 1.

```
LDSMAXL <Xs>, <Xt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LSE) then UNDEFINED;

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

constant integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
boolean acquire = A == '1' && Rt != '11111';
boolean release = R == '1';
boolean tagchecked = n != 31;
```

Alias conditions

Alias	is preferred when
STSMAX, STSMAXL	A == '0' && Rt == '11111'

Assembler symbols

<Ws>	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Wt>	Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xs>	Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Xt>	Is the 64-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(datasize) value;
bits(datasize) data;

AccessDescriptor accdesc = CreateAccDescAtomicOp(MemAtomicOp_SMAX, acquire, release, tagchecked);

value = X[s, datasize];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

bits(datasize) comparevalue = bits(datasize) UNKNOWN;    // Irrelevant when not executing CAS
data = MemAtomic(address, comparevalue, value, accdesc);
```

```
if t != 31 then
    X[t, regsize] = ZeroExtend(data, regsize);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

C6.2.204 LDSMAXB, LDSMAXAB, LDSMAXALB, LDSMAXLB

Atomic signed maximum on byte in memory atomically loads an 8-bit byte from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as signed numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, LDSMAXAB and LDSMAXALB load from memory with acquire semantics.
- LDSMAXLB and LDSMAXALB store to memory with release semantics.
- LDSMAXB has neither acquire nor release semantics.

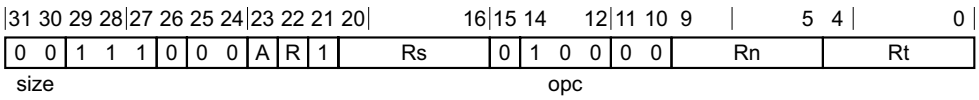
For more information about memory ordering semantics, see *Load-Acquire, Load-AcquirePC, and Store-Release*.

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

This instruction is used by the alias STSMAXB, STSMAXLB. See *Alias conditions* for details of when each alias is preferred.

Integer

(FEAT_LSE)



LDSMAXAB variant

Applies when A == 1 && R == 0.

```
LDSMAXAB <Ws>, <Wt>, [<Xn|SP>]
```

LDSMAXALB variant

Applies when A == 1 && R == 1.

```
LDSMAXALB <Ws>, <Wt>, [<Xn|SP>]
```

LDSMAXB variant

Applies when A == 0 && R == 0.

```
LDSMAXB <Ws>, <Wt>, [<Xn|SP>]
```

LDSMAXLB variant

Applies when A == 0 && R == 1.

```
LDSMAXLB <Ws>, <Wt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LSE) then UNDEFINED;

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

boolean acquire = A == '1' && Rt != '11111';
boolean release = R == '1';
boolean tagchecked = n != 31;
```

Alias conditions

Alias	is preferred when
STSMAXB, STSMAXLB	`A == '0' && Rt == '11111'`

Assembler symbols

`<Ws>` Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

`<Wt>` Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

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

Operation

```
bits(64) address;
bits(8) value;
bits(8) data;

AccessDescriptor accdesc = CreateAccDescAtomicOp(MemAtomicOp_SMAX, acquire, release, tagchecked);

value = X[s, 8];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

bits(8) comparevalue = bits(8) UNKNOWN;    // Irrelevant when not executing CAS
data = MemAtomic(address, comparevalue, value, accdesc);

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

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

C6.2.205 LDSMAXH, LDSMAXAH, LDSMAXALH, LDSMAXLH

Atomic signed maximum on halfword in memory atomically loads a 16-bit halfword from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as signed numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, LDSMAXAH and LDSMAXALH load from memory with acquire semantics.
- LDSMAXLH and LDSMAXALH store to memory with release semantics.
- LDSMAXH has neither acquire nor release semantics.

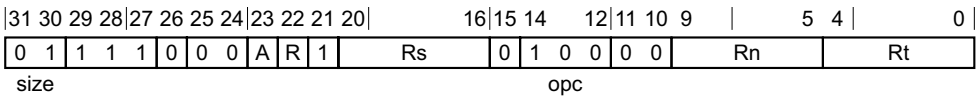
For more information about memory ordering semantics, see *Load-Acquire, Load-AcquirePC, and Store-Release*.

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

This instruction is used by the alias STSMAXH, STSMAXLH. See *Alias conditions* for details of when each alias is preferred.

Integer

(FEAT_LSE)



LDSMAXAH variant

Applies when A == 1 && R == 0.

```
LDSMAXAH <Ws>, <Wt>, [<Xn|SP>]
```

LDSMAXALH variant

Applies when A == 1 && R == 1.

```
LDSMAXALH <Ws>, <Wt>, [<Xn|SP>]
```

LDSMAXH variant

Applies when A == 0 && R == 0.

```
LDSMAXH <Ws>, <Wt>, [<Xn|SP>]
```

LDSMAXLH variant

Applies when A == 0 && R == 1.

```
LDSMAXLH <Ws>, <Wt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LSE) then UNDEFINED;

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

boolean acquire = A == '1' && Rt != '11111';
boolean release = R == '1';
boolean tagchecked = n != 31;
```

Alias conditions

Alias	is preferred when
STSMAXH, STSMAXLH	`A == '0' && Rt == '11111'`

Assembler symbols

`<Ws>` Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

`<Wt>` Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

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

Operation

```
bits(64) address;
bits(16) value;
bits(16) data;

AccessDescriptor accdesc = CreateAccDescAtomicOp(MemAtomicOp_SMAX, acquire, release, tagchecked);

value = X[s, 16];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

bits(16) comparevalue = bits(16) UNKNOWN;    // Irrelevant when not executing CAS
data = MemAtomic(address, comparevalue, value, accdesc);

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

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

C6.2.206 LDSMIN, LDSMINA, LDSMINAL, LDSMINL

Atomic signed minimum on word or doubleword in memory atomically loads a 32-bit word or 64-bit doubleword from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as signed numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not one of `WZR` or `XZR`, `LDSMINA` and `LDSMINAL` load from memory with acquire semantics.
- `LDSMINL` and `LDSMINAL` store to memory with release semantics.
- `LDSMIN` has neither acquire nor release semantics.

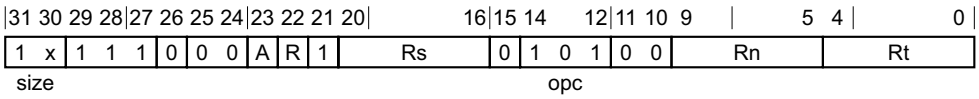
For more information about memory ordering semantics, see *Load-Acquire, Load-AcquirePC, and Store-Release*.

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

This instruction is used by the alias STSMIN, STSMINL. See *Alias conditions* for details of when each alias is preferred.

Integer

(FEAT_LSE)



32-bit LDSMIN variant

Applies when `size == 10 && A == 0 && R == 0`.

```
LDSMIN <Ws>, <Wt>, [<Xn|SP>]
```

32-bit LDSMINA variant

Applies when `size == 10 && A == 1 && R == 0`.

```
LDSMINA <Ws>, <Wt>, [<Xn|SP>]
```

32-bit LDSMINAL variant

Applies when `size == 10 && A == 1 && R == 1`.

```
LDSMINAL <Ws>, <Wt>, [<Xn|SP>]
```

32-bit LDSMINL variant

Applies when `size == 10 && A == 0 && R == 1`.

```
LDSMINL <Ws>, <Wt>, [<Xn|SP>]
```

64-bit LDSMIN variant

Applies when `size == 11 && A == 0 && R == 0`.

```
LDSMIN <Xs>, <Xt>, [<Xn|SP>]
```

64-bit LDSMINA variant

Applies when `size == 11 && A == 1 && R == 0`.

```
LDSMINA <Xs>, <Xt>, [<Xn|SP>]
```

64-bit LDSMINAL variant

Applies when size == 11 && A == 1 && R == 1.

```
LDSMINAL <Xs>, <Xt>, [<Xn|SP>]
```

64-bit LDSMINL variant

Applies when size == 11 && A == 0 && R == 1.

```
LDSMINL <Xs>, <Xt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LSE) then UNDEFINED;

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

constant integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
boolean acquire = A == '1' && Rt != '11111';
boolean release = R == '1';
boolean tagchecked = n != 31;
```

Alias conditions

Alias	is preferred when
STSMIN, STSMINL	A == '0' && Rt == '11111'

Assembler symbols

<Ws>	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Wt>	Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xs>	Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Xt>	Is the 64-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(datasize) value;
bits(datasize) data;

AccessDescriptor accdesc = CreateAccDescAtomicOp(MemAtomicOp_SMIN, acquire, release, tagchecked);

value = X[s, datasize];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

bits(datasize) comparevalue = bits(datasize) UNKNOWN;    // Irrelevant when not executing CAS
data = MemAtomic(address, comparevalue, value, accdesc);
```

```
if t != 31 then
    X[t, regsize] = ZeroExtend(data, regsize);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

C6.2.207 LDSMINB, LDSMINAB, LDSMINALB, LDSMINLB

Atomic signed minimum on byte in memory atomically loads an 8-bit byte from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as signed numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, LDSMINAB and LDSMINALB load from memory with acquire semantics.
- LDSMINLB and LDSMINALB store to memory with release semantics.
- LDSMINB has neither acquire nor release semantics.

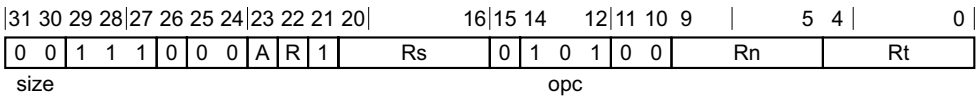
For more information about memory ordering semantics, see *Load-Acquire, Load-AcquirePC, and Store-Release*.

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

This instruction is used by the alias STSMINB, STSMINLB. See *Alias conditions* for details of when each alias is preferred.

Integer

(FEAT_LSE)



LDSMINAB variant

Applies when A == 1 && R == 0.

```
LDSMINAB <Ws>, <Wt>, [<Xn|SP>]
```

LDSMINALB variant

Applies when A == 1 && R == 1.

```
LDSMINALB <Ws>, <Wt>, [<Xn|SP>]
```

LDSMINB variant

Applies when A == 0 && R == 0.

```
LDSMINB <Ws>, <Wt>, [<Xn|SP>]
```

LDSMINLB variant

Applies when A == 0 && R == 1.

```
LDSMINLB <Ws>, <Wt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LSE) then UNDEFINED;

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

boolean acquire = A == '1' && Rt != '11111';
boolean release = R == '1';
boolean tagchecked = n != 31;
```

Alias conditions

Alias	is preferred when
STSMINB, STSMINLB	A == '0' && Rt == '11111'

Assembler symbols

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Wt> Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

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

Operation

```
bits(64) address;
bits(8) value;
bits(8) data;

AccessDescriptor accdesc = CreateAccDescAtomicOp(MemAtomicOp_SMIN, acquire, release, tagchecked);

value = X[s, 8];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

bits(8) comparevalue = bits(8) UNKNOWN;    // Irrelevant when not executing CAS
data = MemAtomic(address, comparevalue, value, accdesc);

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

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

C6.2.208 LDSMINH, LDSMINAH, LDSMINALH, LDSMINLH

Atomic signed minimum on halfword in memory atomically loads a 16-bit halfword from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as signed numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, LDSMINAH and LDSMINALH load from memory with acquire semantics.
- LDSMINLH and LDSMINALH store to memory with release semantics.
- LDSMINH has neither acquire nor release semantics.

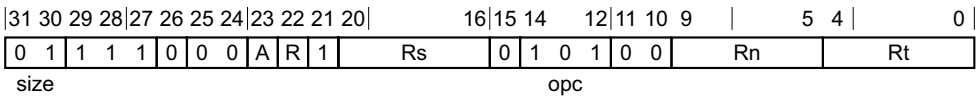
For more information about memory ordering semantics, see *Load-Acquire, Load-AcquirePC, and Store-Release*.

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

This instruction is used by the alias STSMINH, STSMINLH. See *Alias conditions* for details of when each alias is preferred.

Integer

(FEAT_LSE)



LDSMINAH variant

Applies when A == 1 && R == 0.

```
LDSMINAH <Ws>, <Wt>, [<Xn|SP>]
```

LDSMINALH variant

Applies when A == 1 && R == 1.

```
LDSMINALH <Ws>, <Wt>, [<Xn|SP>]
```

LDSMINH variant

Applies when A == 0 && R == 0.

```
LDSMINH <Ws>, <Wt>, [<Xn|SP>]
```

LDSMINLH variant

Applies when A == 0 && R == 1.

```
LDSMINLH <Ws>, <Wt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LSE) then UNDEFINED;

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

boolean acquire = A == '1' && Rt != '11111';
boolean release = R == '1';
boolean tagchecked = n != 31;
```

Alias conditions

Alias	is preferred when
STSMINH, STSMINLH	A == '0' && Rt == '11111'

Assembler symbols

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Wt> Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

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

Operation

```
bits(64) address;
bits(16) value;
bits(16) data;

AccessDescriptor accdesc = CreateAccDescAtomicOp(MemAtomicOp_SMIN, acquire, release, tagchecked);

value = X[s, 16];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

bits(16) comparevalue = bits(16) UNKNOWN;    // Irrelevant when not executing CAS
data = MemAtomic(address, comparevalue, value, accdesc);

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

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

C6.2.209 LDTR

Load Register (unprivileged) loads a word or doubleword from memory, and writes it to a register. The address that is used for the load is calculated from a base register and an immediate offset.

Memory accesses made by the instruction behave as if the instruction was executed at EL0 if the *Effective value* of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the *Effective value* of HCR_EL2.{E2H, TGE} is {1, 1}.

Otherwise, the memory access operates with the restrictions determined by the Exception level at which the instruction is executed. For information about memory accesses, see *Load/store addressing modes*.

32-bit variant

Applies when `size == 10`.

```
LDTR <Wt>, [<Xn|SP>{, #<simm>}]
```

64-bit variant

Applies when `size == 11`.

```
LDTR <Xt>, [<Xn|SP>{, #<simm>}]
```

Decode for all variants of this encoding

```
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

Assembler symbols

`<Wt>`	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
`<Xt>`	Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
`<Xn|SP>`	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
`<simm>`	Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared decode for all encodings

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer regsize;

regsize = if size == '11' then 64 else 32;
constant integer datasize = 8 << scale;
boolean tagchecked = n != 31;
```

Operation

```
bits(64) address;
bits(datasize) data;

boolean privileged = AArch64.IsUnprivAccessPriv();
AccessDescriptor accdesc = CreateAccDescGPR(MemOp_LOAD, FALSE, privileged, tagchecked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

address = GenerateAddress(address, offset, accdesc);

data = Mem[address, datasize DIV 8, accdesc];
X[t, regsize] = ZeroExtend(data, regsize);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

C6.2.210 LDTRB

Load Register Byte (unprivileged) loads a byte from memory, zero-extends it, and writes the result to a register. The address that is used for the load is calculated from a base register and an immediate offset.

Memory accesses made by the instruction behave as if the instruction was executed at EL0 if the *Effective value* of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the *Effective value* of HCR_EL2.{E2H, TGE} is {1, 1}.

Otherwise, the memory access operates with the restrictions determined by the Exception level at which the instruction is executed. For information about memory accesses, see *Load/store addressing modes*.

Encoding

```
LDTRB <Wt>, [<Xn|SP>{, #<simm>}]
```

Decode for this encoding

```
bits(64) offset = SignExtend(imm9, 64);
```

Assembler symbols

<Wt>	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<simm>	Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared decode for all encodings

```
integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tagchecked = n != 31;
```

Operation

```
bits(64) address;
bits(8) data;

boolean privileged = AArch64.IsUnprivAccessPriv();
AccessDescriptor accdesc = CreateAccDescGPR(MemOp_LOAD, FALSE, privileged, tagchecked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

address = GenerateAddress(address, offset, accdesc);
```

```
data = Mem[address, 1, accdesc];
X[t, 32] = ZeroExtend(data, 32);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

C6.2.211 LDTRH

Load Register Halfword (unprivileged) loads a halfword from memory, zero-extends it, and writes the result to a register. The address that is used for the load is calculated from a base register and an immediate offset.

Memory accesses made by the instruction behave as if the instruction was executed at EL0 if the *Effective value* of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the *Effective value* of HCR_EL2.{E2H, TGE} is {1, 1}.

Otherwise, the memory access operates with the restrictions determined by the Exception level at which the instruction is executed. For information about memory accesses, see *Load/store addressing modes*.

Encoding

```
LDTRH <Wt>, [<Xn|SP>{, #<simm>}]
```

Decode for this encoding

```
bits(64) offset = SignExtend(imm9, 64);
```

Assembler symbols

<Wt>	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<simm>	Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared decode for all encodings

```
integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tagchecked = n != 31;
```

Operation

```
bits(64) address;
bits(16) data;

boolean privileged = AArch64.IsUnprivAccessPriv();
AccessDescriptor accdesc = CreateAccDescGPR(MemOp_LOAD, FALSE, privileged, tagchecked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

address = GenerateAddress(address, offset, accdesc);
```

```
data = Mem[address, 2, accdesc];
X[t, 32] = ZeroExtend(data, 32);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

C6.2.212 LDTRSB

Load Register Signed Byte (unprivileged) loads a byte from memory, sign-extends it to 32 bits or 64 bits, and writes the result to a register. The address that is used for the load is calculated from a base register and an immediate offset.

Memory accesses made by the instruction behave as if the instruction was executed at EL0 if the *Effective value* of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the *Effective value* of HCR_EL2.{E2H, TGE} is {1, 1}.

Otherwise, the memory access operates with the restrictions determined by the Exception level at which the instruction is executed. For information about memory accesses, see *Load/store addressing modes*.

32-bit variant

Applies when opc == 11.

```
LDTRSB <Wt>, [<Xn|SP>{, #<simm>}]
```

64-bit variant

Applies when opc == 10.

```
LDTRSB <Xt>, [<Xn|SP>{, #<simm>}]
```

Decode for all variants of this encoding

```
bits(64) offset = SignExtend(imm9, 64);
```

Assembler symbols

<Wt>	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xt>	Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<simm>	Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared decode for all encodings

```
integer n = UInt(Rn);
integer t = UInt(Rt);
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = 32;
    signed = FALSE;
else
```

```
    // sign-extending load
    memop = MemOp_LOAD;
    regsize = if opc<0> == '1' then 32 else 64;
    signed = TRUE;

boolean tagchecked = memop != MemOp_PREFETCH && (n != 31);
```

Operation

```
bits(64) address;
bits(8) data;

boolean privileged = AArch64.IsUnprivAccessPriv();
AccessDescriptor accdesc = CreateAccDescGPR(memop, FALSE, privileged, tagchecked);

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

address = GenerateAddress(address, offset, accdesc);

case memop of
    when MemOp_STORE
        data = X[t, 8];
        Mem[address, 1, accdesc] = data;

    when MemOp_LOAD
        data = Mem[address, 1, accdesc];
        if signed then
            X[t, regsize] = SignExtend(data, regsize);
        else
            X[t, regsize] = ZeroExtend(data, regsize);

    when MemOp_PREFETCH
        Prefetch(address, t<4:0>);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

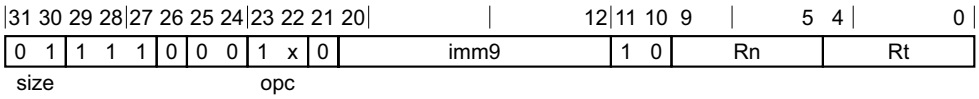
C6.2.213 LDTRSH

Load Register Signed Halfword (unprivileged) loads a halfword from memory, sign-extends it to 32 bits or 64 bits, and writes the result to a register. The address that is used for the load is calculated from a base register and an immediate offset.

Memory accesses made by the instruction behave as if the instruction was executed at EL0 if the *Effective value* of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the *Effective value* of HCR_EL2.{E2H, TGE} is {1, 1}.

Otherwise, the memory access operates with the restrictions determined by the Exception level at which the instruction is executed. For information about memory accesses, see *Load/store addressing modes*.



32-bit variant

Applies when opc == 11.

```
LDTRSH <Wt>, [<Xn|SP>{, #<simm>}]
```

64-bit variant

Applies when opc == 10.

```
LDTRSH <Xt>, [<Xn|SP>{, #<simm>}]
```

Decode for all variants of this encoding

```
bits(64) offset = SignExtend(imm9, 64);
```

Assembler symbols

<Wt>	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xt>	Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<simm>	Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared decode for all encodings

```
integer n = UInt(Rn);
integer t = UInt(Rt);
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = 32;
    signed = FALSE;
```

```
else
    // sign-extending load
    memop = MemOp_LOAD;
    regsize = if opc<0> == '1' then 32 else 64;
    signed = TRUE;

boolean tagchecked = memop != MemOp_PREFETCH && (n != 31);
```

Operation

```
bits(64) address;
bits(16) data;

boolean privileged = AArch64.IsUnprivAccessPriv();
AccessDescriptor accdesc = CreateAccDescGPR(memop, FALSE, privileged, tagchecked);

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

address = GenerateAddress(address, offset, accdesc);

case memop of
    when MemOp_STORE
        data = X[t, 16];
        Mem[address, 2, accdesc] = data;

    when MemOp_LOAD
        data = Mem[address, 2, accdesc];
        if signed then
            X[t, regsize] = SignExtend(data, regsize);
        else
            X[t, regsize] = ZeroExtend(data, regsize);

    when MemOp_PREFETCH
        Prefetch(address, t<4:0>);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

C6.2.214 LDTRSW

Load Register Signed Word (unprivileged) loads a word from memory, sign-extends it to 64 bits, and writes the result to a register. The address that is used for the load is calculated from a base register and an immediate offset.

Memory accesses made by the instruction behave as if the instruction was executed at EL0 if the *Effective value* of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the *Effective value* of HCR_EL2.{E2H, TGE} is {1, 1}.

Otherwise, the memory access operates with the restrictions determined by the Exception level at which the instruction is executed. For information about memory accesses, see *Load/store addressing modes*.

Encoding

```
LDTRSW <Xt>, [<Xn|SP>{, #<simm>}]
```

Decode for this encoding

```
bits(64) offset = SignExtend(imm9, 64);
```

Assembler symbols

<Xt>	Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<simm>	Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared decode for all encodings

```
integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tagchecked = n != 31;
```

Operation

```
bits(64) address;
bits(32) data;

boolean privileged = AArch64.IsUnprivAccessPriv();
AccessDescriptor accdesc = CreateAccDescGPR(MemOp_LOAD, FALSE, privileged, tagchecked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

address = GenerateAddress(address, offset, accdesc);
```

```
data = Mem[address, 4, accdesc];
X[t, 64] = SignExtend(data, 64);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

C6.2.215 LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL

Atomic unsigned maximum on word or doubleword in memory atomically loads a 32-bit word or 64-bit doubleword from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as unsigned numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not one of `WZR` or `XZR`, `LDUMAXA` and `LDUMAXAL` load from memory with acquire semantics.
- `LDUMAXL` and `LDUMAXAL` store to memory with release semantics.
- `LDUMAX` has neither acquire nor release semantics.

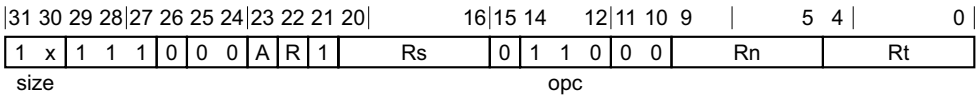
For more information about memory ordering semantics, see *Load-Acquire, Load-AcquirePC, and Store-Release*.

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

This instruction is used by the alias STUMAX, STUMAXL. See *Alias conditions* for details of when each alias is preferred.

Integer

(FEAT_LSE)



32-bit LDUMAX variant

Applies when `size == 10 && A == 0 && R == 0`.

```
LDUMAX <Ws>, <Wt>, [<Xn|SP>]
```

32-bit LDUMAXA variant

Applies when `size == 10 && A == 1 && R == 0`.

```
LDUMAXA <Ws>, <Wt>, [<Xn|SP>]
```

32-bit LDUMAXAL variant

Applies when `size == 10 && A == 1 && R == 1`.

```
LDUMAXAL <Ws>, <Wt>, [<Xn|SP>]
```

32-bit LDUMAXL variant

Applies when `size == 10 && A == 0 && R == 1`.

```
LDUMAXL <Ws>, <Wt>, [<Xn|SP>]
```

64-bit LDUMAX variant

Applies when `size == 11 && A == 0 && R == 0`.

```
LDUMAX <Xs>, <Xt>, [<Xn|SP>]
```

64-bit LDUMAXA variant

Applies when `size == 11 && A == 1 && R == 0`.

```
LDUMAXA <Xs>, <Xt>, [<Xn|SP>]
```

64-bit LDUMAXAL variant

Applies when size == 11 && A == 1 && R == 1.

```
LDUMAXAL <Xs>, <Xt>, [<Xn|SP>]
```

64-bit LDUMAXL variant

Applies when size == 11 && A == 0 && R == 1.

```
LDUMAXL <Xs>, <Xt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LSE) then UNDEFINED;

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

constant integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
boolean acquire = A == '1' && Rt != '11111';
boolean release = R == '1';
boolean tagchecked = n != 31;
```

Alias conditions

Alias	is preferred when
STUMAX, STUMAXL	A == '0' && Rt == '11111'

Assembler symbols

<Ws>	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Wt>	Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xs>	Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Xt>	Is the 64-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(datasize) value;
bits(datasize) data;

AccessDescriptor accdesc = CreateAccDescAtomicOp(MemAtomicOp_UMAX, acquire, release, tagchecked);

value = X[s, datasize];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

bits(datasize) comparevalue = bits(datasize) UNKNOWN;    // Irrelevant when not executing CAS
data = MemAtomic(address, comparevalue, value, accdesc);
```

```
if t != 31 then
    X[t, regsize] = ZeroExtend(data, regsize);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

C6.2.216 LDUMAXB, LDUMAXAB, LDUMAXALB, LDUMAXLB

Atomic unsigned maximum on byte in memory atomically loads an 8-bit byte from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as unsigned numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, LDUMAXAB and LDUMAXALB load from memory with acquire semantics.
- LDUMAXLB and LDUMAXALB store to memory with release semantics.
- LDUMAXB has neither acquire nor release semantics.

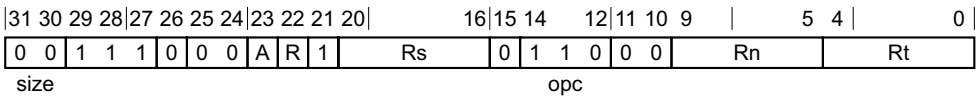
For more information about memory ordering semantics, see *Load-Acquire, Load-AcquirePC, and Store-Release*.

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

This instruction is used by the alias STUMAXB, STUMAXLB. See *Alias conditions* for details of when each alias is preferred.

Integer

(FEAT_LSE)



LDUMAXAB variant

Applies when A == 1 && R == 0.

```
LDUMAXAB <Ws>, <Wt>, [<Xn|SP>]
```

LDUMAXALB variant

Applies when A == 1 && R == 1.

```
LDUMAXALB <Ws>, <Wt>, [<Xn|SP>]
```

LDUMAXB variant

Applies when A == 0 && R == 0.

```
LDUMAXB <Ws>, <Wt>, [<Xn|SP>]
```

LDUMAXLB variant

Applies when A == 0 && R == 1.

```
LDUMAXLB <Ws>, <Wt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LSE) then UNDEFINED;

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

boolean acquire = A == '1' && Rt != '11111';
boolean release = R == '1';
boolean tagchecked = n != 31;
```

Alias conditions

Alias	is preferred when
STUMAXB, STUMAXLB	`A == '0' && Rt == '11111'`

Assembler symbols

`<Ws>` Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

`<Wt>` Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

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

Operation

```
bits(64) address;
bits(8) value;
bits(8) data;

AccessDescriptor accdesc = CreateAccDescAtomicOp(MemAtomicOp_UMAX, acquire, release, tagchecked);

value = X[s, 8];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

bits(8) comparevalue = bits(8) UNKNOWN;    // Irrelevant when not executing CAS
data = MemAtomic(address, comparevalue, value, accdesc);

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

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

C6.2.217 LDUMAXH, LDUMAXAH, LDUMAXALH, LDUMAXLH

Atomic unsigned maximum on halfword in memory atomically loads a 16-bit halfword from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as unsigned numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, LDUMAXAH and LDUMAXALH load from memory with acquire semantics.
- LDUMAXLH and LDUMAXALH store to memory with release semantics.
- LDUMAXH has neither acquire nor release semantics.

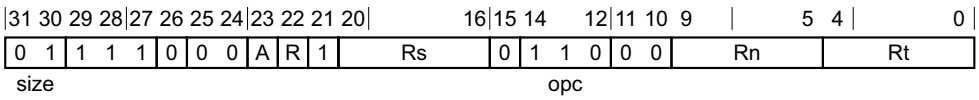
For more information about memory ordering semantics, see *Load-Acquire, Load-AcquirePC, and Store-Release*.

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

This instruction is used by the alias STUMAXH, STUMAXLH. See *Alias conditions* for details of when each alias is preferred.

Integer

(FEAT_LSE)



LDUMAXAH variant

Applies when A == 1 && R == 0.

```
LDUMAXAH <Ws>, <Wt>, [<Xn|SP>]
```

LDUMAXALH variant

Applies when A == 1 && R == 1.

```
LDUMAXALH <Ws>, <Wt>, [<Xn|SP>]
```

LDUMAXH variant

Applies when A == 0 && R == 0.

```
LDUMAXH <Ws>, <Wt>, [<Xn|SP>]
```

LDUMAXLH variant

Applies when A == 0 && R == 1.

```
LDUMAXLH <Ws>, <Wt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LSE) then UNDEFINED;

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

boolean acquire = A == '1' && Rt != '11111';
boolean release = R == '1';
boolean tagchecked = n != 31;
```

Alias conditions

Alias	is preferred when
STUMAXH, STUMAXLH	`A == '0' && Rt == '11111'`

Assembler symbols

`<Ws>` Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

`<Wt>` Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

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

Operation

```
bits(64) address;
bits(16) value;
bits(16) data;

AccessDescriptor accdesc = CreateAccDescAtomicOp(MemAtomicOp_UMAX, acquire, release, tagchecked);

value = X[s, 16];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

bits(16) comparevalue = bits(16) UNKNOWN;    // Irrelevant when not executing CAS
data = MemAtomic(address, comparevalue, value, accdesc);

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

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

C6.2.218 LDUMIN, LDUMINA, LDUMINAL, LDUMINL

Atomic unsigned minimum on word or doubleword in memory atomically loads a 32-bit word or 64-bit doubleword from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as unsigned numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not one of `WZR` or `XZR`, `LDUMINA` and `LDUMINAL` load from memory with acquire semantics.
- `LDUMINL` and `LDUMINAL` store to memory with release semantics.
- `LDUMIN` has neither acquire nor release semantics.

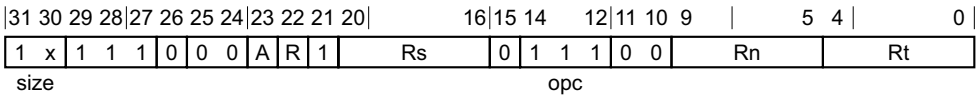
For more information about memory ordering semantics, see *Load-Acquire, Load-AcquirePC, and Store-Release*.

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

This instruction is used by the alias STUMIN, STUMINL. See *Alias conditions* for details of when each alias is preferred.

Integer

(FEAT_LSE)



32-bit LDUMIN variant

Applies when `size == 10 && A == 0 && R == 0`.

```
LDUMIN <Ws>, <Wt>, [<Xn|SP>]
```

32-bit LDUMINA variant

Applies when `size == 10 && A == 1 && R == 0`.

```
LDUMINA <Ws>, <Wt>, [<Xn|SP>]
```

32-bit LDUMINAL variant

Applies when `size == 10 && A == 1 && R == 1`.

```
LDUMINAL <Ws>, <Wt>, [<Xn|SP>]
```

32-bit LDUMINL variant

Applies when `size == 10 && A == 0 && R == 1`.

```
LDUMINL <Ws>, <Wt>, [<Xn|SP>]
```

64-bit LDUMIN variant

Applies when `size == 11 && A == 0 && R == 0`.

```
LDUMIN <Xs>, <Xt>, [<Xn|SP>]
```

64-bit LDUMINA variant

Applies when `size == 11 && A == 1 && R == 0`.

```
LDUMINA <Xs>, <Xt>, [<Xn|SP>]
```

64-bit LDUMINAL variant

Applies when size == 11 && A == 1 && R == 1.

```
LDUMINAL <Xs>, <Xt>, [<Xn|SP>]
```

64-bit LDUMINL variant

Applies when size == 11 && A == 0 && R == 1.

```
LDUMINL <Xs>, <Xt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LSE) then UNDEFINED;

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

constant integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
boolean acquire = A == '1' && Rt != '11111';
boolean release = R == '1';
boolean tagchecked = n != 31;
```

Alias conditions

Alias	is preferred when
STUMIN, STUMINL	A == '0' && Rt == '11111'

Assembler symbols

<Ws>	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Wt>	Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xs>	Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Xt>	Is the 64-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(datasize) value;
bits(datasize) data;

AccessDescriptor accdesc = CreateAccDescAtomicOp(MemAtomicOp_UMIN, acquire, release, tagchecked);

value = X[s, datasize];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

bits(datasize) comparevalue = bits(datasize) UNKNOWN;    // Irrelevant when not executing CAS
data = MemAtomic(address, comparevalue, value, accdesc);
```

```
if t != 31 then
    X[t, regsize] = ZeroExtend(data, regsize);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

C6.2.219 LDUMINB, LDUMINAB, LDUMINALB, LDUMINLB

Atomic unsigned minimum on byte in memory atomically loads an 8-bit byte from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as unsigned numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, LDUMINAB and LDUMINALB load from memory with acquire semantics.
- LDUMINLB and LDUMINALB store to memory with release semantics.
- LDUMINB has neither acquire nor release semantics.

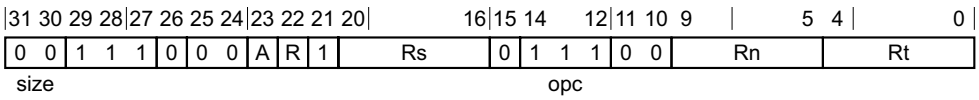
For more information about memory ordering semantics, see *Load-Acquire, Load-AcquirePC, and Store-Release*.

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

This instruction is used by the alias STUMINB, STUMINLB. See *Alias conditions* for details of when each alias is preferred.

Integer

(FEAT_LSE)



LDUMINAB variant

Applies when A == 1 && R == 0.

```
LDUMINAB <Ws>, <Wt>, [<Xn|SP>]
```

LDUMINALB variant

Applies when A == 1 && R == 1.

```
LDUMINALB <Ws>, <Wt>, [<Xn|SP>]
```

LDUMINB variant

Applies when A == 0 && R == 0.

```
LDUMINB <Ws>, <Wt>, [<Xn|SP>]
```

LDUMINLB variant

Applies when A == 0 && R == 1.

```
LDUMINLB <Ws>, <Wt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LSE) then UNDEFINED;

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

boolean acquire = A == '1' && Rt != '11111';
boolean release = R == '1';
boolean tagchecked = n != 31;
```

Alias conditions

Alias	is preferred when
STUMINB, STUMINLB	`A == '0' && Rt == '11111'`

Assembler symbols

`<Ws>` Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

`<Wt>` Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

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

Operation

```
bits(64) address;
bits(8) value;
bits(8) data;

AccessDescriptor accdesc = CreateAccDescAtomicOp(MemAtomicOp_UMIN, acquire, release, tagchecked);

value = X[s, 8];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

bits(8) comparevalue = bits(8) UNKNOWN;    // Irrelevant when not executing CAS
data = MemAtomic(address, comparevalue, value, accdesc);

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

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

C6.2.220 LDUMINH, LDUMINAH, LDUMINALH, LDUMINLH

Atomic unsigned minimum on halfword in memory atomically loads a 16-bit halfword from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as unsigned numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, LDUMINAH and LDUMINALH load from memory with acquire semantics.
- LDUMINLH and LDUMINALH store to memory with release semantics.
- LDUMINH has neither acquire nor release semantics.

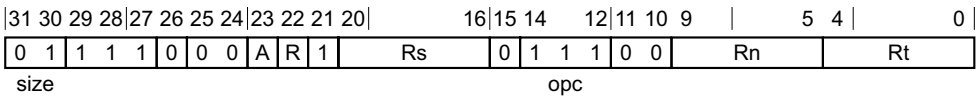
For more information about memory ordering semantics, see *Load-Acquire, Load-AcquirePC, and Store-Release*.

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

This instruction is used by the alias STUMINH, STUMINLH. See *Alias conditions* for details of when each alias is preferred.

Integer

(FEAT_LSE)



LDUMINAH variant

Applies when A == 1 && R == 0.

```
LDUMINAH <Ws>, <Wt>, [<Xn|SP>]
```

LDUMINALH variant

Applies when A == 1 && R == 1.

```
LDUMINALH <Ws>, <Wt>, [<Xn|SP>]
```

LDUMINH variant

Applies when A == 0 && R == 0.

```
LDUMINH <Ws>, <Wt>, [<Xn|SP>]
```

LDUMINLH variant

Applies when A == 0 && R == 1.

```
LDUMINLH <Ws>, <Wt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LSE) then UNDEFINED;

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

boolean acquire = A == '1' && Rt != '11111';
boolean release = R == '1';
boolean tagchecked = n != 31;
```

Alias conditions

Alias	is preferred when
STUMINH, STUMINLH	`A == '0' && Rt == '11111'`

Assembler symbols

`<Ws>` Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

`<Wt>` Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

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

Operation

```
bits(64) address;
bits(16) value;
bits(16) data;

AccessDescriptor accdesc = CreateAccDescAtomicOp(MemAtomicOp_UMIN, acquire, release, tagchecked);

value = X[s, 16];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

bits(16) comparevalue = bits(16) UNKNOWN;    // Irrelevant when not executing CAS
data = MemAtomic(address, comparevalue, value, accdesc);

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

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

C6.2.221 LDUR

Load Register (unscaled) calculates an address from a base register and an immediate offset, loads a 32-bit word or 64-bit doubleword from memory, zero-extends it, and writes it to a register. For information about memory accesses, see *Load/store addressing modes*.

32-bit variant

Applies when `size == 10`.

```
LDUR <Wt>, [<Xn|SP>{, #<simm>}]
```

64-bit variant

Applies when `size == 11`.

```
LDUR <Xt>, [<Xn|SP>{, #<simm>}]
```

Decode for all variants of this encoding

```
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

Assembler symbols

`<Wt>`	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
`<Xt>`	Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
`<Xn|SP>`	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
`<simm>`	Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared decode for all encodings

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer regsize;

regsize = if size == '11' then 64 else 32;
constant integer datasize = 8 << scale;
boolean tagchecked = n != 31;
```

Operation

```
bits(64) address;
bits(datasize) data;

boolean privileged = PSTATE.EL != EL0;
AccessDescriptor accdesc = CreateAccDescGPR(MemOp_LOAD, FALSE, privileged, tagchecked);

if n == 31 then
    CheckSPAlignment();
```

```
    address = SP[];
else
    address = X[n, 64];

address = GenerateAddress(address, offset, accdesc);

data = Mem[address, datasize DIV 8, accdesc];
X[t, regsize] = ZeroExtend(data, regsize);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

C6.2.222 LDURB

Load Register Byte (unscaled) calculates an address from a base register and an immediate offset, loads a byte from memory, zero-extends it, and writes it to a register. For information about memory accesses, see *Load/store addressing modes*.

Encoding

```
LDURB <Wt>, [<Xn|SP>{, #<simm>}]
```

Decode for this encoding

```
bits(64) offset = SignExtend(imm9, 64);
```

Assembler symbols

<Wt>	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<simm>	Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared decode for all encodings

```
integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tagchecked = n != 31;
```

Operation

```
bits(64) address;
bits(8) data;

boolean privileged = PSTATE.EL != EL0;
AccessDescriptor accdesc = CreateAccDescGPR(MemOp_LOAD, FALSE, privileged, tagchecked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

address = GenerateAddress(address, offset, accdesc);

data = Mem[address, 1, accdesc];
X[t, 32] = ZeroExtend(data, 32);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

C6.2.223 LDURH

Load Register Halfword (unscaled) calculates an address from a base register and an immediate offset, loads a halfword from memory, zero-extends it, and writes it to a register. For information about memory accesses, see *Load/store addressing modes*.

Encoding

```
LDURH <Wt>, [<Xn|SP>{, #<simm>}]
```

Decode for this encoding

```
bits(64) offset = SignExtend(imm9, 64);
```

Assembler symbols

<Wt>	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<simm>	Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared decode for all encodings

```
integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tagchecked = n != 31;
```

Operation

```
bits(64) address;
bits(16) data;

boolean privileged = PSTATE.EL != EL0;
AccessDescriptor accdesc = CreateAccDescGPR(MemOp_LOAD, FALSE, privileged, tagchecked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

address = GenerateAddress(address, offset, accdesc);

data = Mem[address, 2, accdesc];
X[t, 32] = ZeroExtend(data, 32);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

C6.2.224 LDURSB

Load Register Signed Byte (unscaled) calculates an address from a base register and an immediate offset, loads a signed byte from memory, sign-extends it, and writes it to a register. For information about memory accesses, see *Load/store addressing modes*.

32-bit variant

Applies when opc == 11.

```
LDURSB <Wt>, [<Xn|SP>{, #<simm>}]
```

64-bit variant

Applies when opc == 10.

```
LDURSB <Xt>, [<Xn|SP>{, #<simm>}]
```

Decode for all variants of this encoding

```
bits(64) offset = SignExtend(imm9, 64);
```

Assembler symbols

<Wt>	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xt>	Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<simm>	Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared decode for all encodings

```
integer n = UInt(Rn);
integer t = UInt(Rt);
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = 32;
    signed = FALSE;
else
    // sign-extending load
    memop = MemOp_LOAD;
    regsize = if opc<0> == '1' then 32 else 64;
    signed = TRUE;

boolean tagchecked = memop != MemOp_PREFETCH && (n != 31);
```

Operation

```
bits(64) address;
bits(8) data;

boolean privileged = PSTATE.EL != EL0;
AccessDescriptor accdesc = CreateAccDescGPR(memop, FALSE, privileged, tagchecked);

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

address = GenerateAddress(address, offset, accdesc);

case memop of
    when MemOp_STORE
        data = X[t, 8];
        Mem[address, 1, accdesc] = data;

    when MemOp_LOAD
        data = Mem[address, 1, accdesc];
        if signed then
            X[t, regsize] = SignExtend(data, regsize);
        else
            X[t, regsize] = ZeroExtend(data, regsize);

    when MemOp_PREFETCH
        Prefetch(address, t<4:0>);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

C6.2.225 LDURSH

Load Register Signed Halfword (unscaled) calculates an address from a base register and an immediate offset, loads a signed halfword from memory, sign-extends it, and writes it to a register. For information about memory accesses, see *Load/store addressing modes*.

32-bit variant

Applies when opc == 11.

```
LDURSH <Wt>, [<Xn|SP>{, #<simm>}]
```

64-bit variant

Applies when opc == 10.

```
LDURSH <Xt>, [<Xn|SP>{, #<simm>}]
```

Decode for all variants of this encoding

```
bits(64) offset = SignExtend(imm9, 64);
```

Assembler symbols

<Wt>	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xt>	Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<simm>	Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared decode for all encodings

```
integer n = UInt(Rn);
integer t = UInt(Rt);
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = 32;
    signed = FALSE;
else
    // sign-extending load
    memop = MemOp_LOAD;
    regsize = if opc<0> == '1' then 32 else 64;
    signed = TRUE;

boolean tagchecked = memop != MemOp_PREFETCH && (n != 31);
```

Operation

```
bits(64) address;
bits(16) data;

boolean privileged = PSTATE.EL != EL0;
AccessDescriptor accdesc = CreateAccDescGPR(memop, FALSE, privileged, tagchecked);

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

address = GenerateAddress(address, offset, accdesc);

case memop of
    when MemOp_STORE
        data = X[t, 16];
        Mem[address, 2, accdesc] = data;

    when MemOp_LOAD
        data = Mem[address, 2, accdesc];
        if signed then
            X[t, regsize] = SignExtend(data, regsize);
        else
            X[t, regsize] = ZeroExtend(data, regsize);

    when MemOp_PREFETCH
        Prefetch(address, t<4:0>);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

C6.2.226 LDURSW

Load Register Signed Word (unscaled) calculates an address from a base register and an immediate offset, loads a signed word from memory, sign-extends it, and writes it to a register. For information about memory accesses, see *Load/store addressing modes*.

Encoding

```
LDURSW <Xt>, [<Xn|SP>{, #<simm>}]
```

Decode for this encoding

```
bits(64) offset = SignExtend(imm9, 64);
```

Assembler symbols

<Xt>	Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<simm>	Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared decode for all encodings

```
integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tagchecked = n != 31;
```

Operation

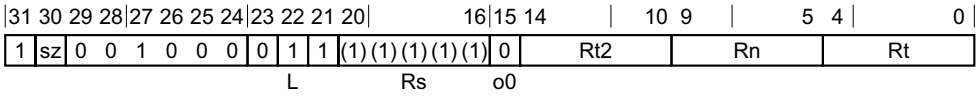
```
bits(64) address;
bits(32) data;

boolean privileged = PSTATE.EL != EL0;
AccessDescriptor accdesc = CreateAccDescGPR(MemOp_LOAD, FALSE, privileged, tagchecked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

address = GenerateAddress(address, offset, accdesc);

data = Mem[address, 4, accdesc];
X[t, 64] = SignExtend(data, 64);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

C6.2.227 LDXP

Load Exclusive Pair of Registers derives an address from a base register value, loads two 32-bit words or two 64-bit doublewords from memory, and writes them to two registers. For information on single-copy atomicity and alignment requirements, see *Requirements for single-copy atomicity* and *Alignment of data accesses*. The PE marks the physical address being accessed as an exclusive access. This exclusive access mark is checked by Store Exclusive instructions. See *Synchronization and semaphores*. For information about memory accesses, see *Load/store addressing modes*.



32-bit variant

Applies when sz == 0.

```
LDXP <Wt1>, <Wt2>, [<Xn|SP>{,#0}]
```

64-bit variant

Applies when sz == 1.

```
LDXP <Xt1>, <Xt2>, [<Xn|SP>{,#0}]
```

Decode for all variants of this encoding

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2);

constant integer elsize = 32 << UInt(sz);
constant integer datasize = elsize * 2;
boolean tagchecked = n != 31;

boolean rt_unknown = FALSE;
if t == t2 then
    Constraint c = ConstrainUnpredictable(Unpredictable_LDPOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN rt_unknown = TRUE;    // result is UNKNOWN
        when Constraint_UNDEF   UNDEFINED;
        when Constraint_NOP     EndOfInstruction();
```

Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*, and particularly *LDXP*.

Assembler symbols

<Wt1> Is the 32-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.

<Wt2> Is the 32-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.

<Xt1> Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.

<Xt2> Is the 64-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.

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

Operation

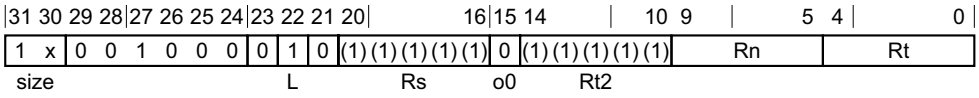
```
bits(64) address;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;

AccessDescriptor accdesc = CreateAccDescExLDST(MemOp_LOAD, FALSE, tagchecked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

// Tell the Exclusives monitors to record a sequence of one or more atomic
// memory reads from virtual address range [address, address+dbytes-1].
// The Exclusives monitor will only be set if all the reads are from the
// same dbytes-aligned physical address, to allow for the possibility of
// an atomicity break if the translation is changed between reads.
AArch64.SetExclusiveMonitors(address, dbytes);

if rt_unknown then
    // ConstrainedUNPREDICTABLE case
    X[t, datasize] = bits(datasize) UNKNOWN;    // In this case t = t2
elsif elsize == 32 then
    // 32-bit load exclusive pair (atomic)
    data = Mem[address, dbytes, accdesc];
    if BigEndian(accdesc.acctype) then
        X[t, datasize-elsize] = data<datasize-1:elsize>;
        X[t2, elsize] = data<elsize-1:0>;
    else
        X[t, elsize] = data<elsize-1:0>;
        X[t2, datasize-elsize] = data<datasize-1:elsize>;
else // elsize == 64
    // 64-bit load exclusive pair (not atomic), but must be 128-bit aligned
    if !IsAligned(address, dbytes) then
        AArch64.Abort(address, AlignmentFault(accdesc));

    bits(64) address2 = GenerateAddress(address, 8, accdesc);
    X[t, 64] = Mem[address, 8, accdesc];
    X[t2, 64] = Mem[address2, 8, accdesc];
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

C6.2.228 LDXR

Load Exclusive Register derives an address from a base register value, loads a 32-bit word or a 64-bit doubleword from memory, and writes it to a register. The memory access is atomic. The PE marks the physical address being accessed as an exclusive access. This exclusive access mark is checked by Store Exclusive instructions. See *Synchronization and semaphores*. For information about memory accesses, see *Load/store addressing modes*.



32-bit variant

Applies when `size == 10`.

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

64-bit variant

Applies when `size == 11`.

```
LDXR <Xt>, [<Xn|SP>{,#0}]
```

Decode for all variants of this encoding

```
integer n = UInt(Rn);
integer t = UInt(Rt);

constant integer elsize = 8 << UInt(size);
integer regsize = if elsize == 64 then 64 else 32;
boolean tagchecked = n != 31;
```

Assembler symbols

`<Wt>`	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
`<Xt>`	Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
`<Xn|SP>`	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(elsize) data;
constant integer dbytes = elsize DIV 8;

AccessDescriptor accdesc = CreateAccDescExLDST(MemOp_LOAD, FALSE, tagchecked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

// Tell the Exclusives monitors to record a sequence of one or more atomic
// memory reads from virtual address range [address, address+dbytes-1].
// The Exclusives monitor will only be set if all the reads are from the
// same dbytes-aligned physical address, to allow for the possibility of
// an atomicity break if the translation is changed between reads.
AArch64.SetExclusiveMonitors(address, dbytes);
```

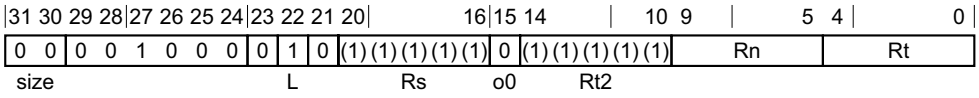
```
data = Mem[address, dbytes, accdesc];
X[t, regsize] = ZeroExtend(data, regsize);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

C6.2.229 LDXRB

Load Exclusive Register Byte derives an address from a base register value, loads a byte from memory, zero-extends it and writes it to a register. The memory access is atomic. The PE marks the physical address being accessed as an exclusive access. This exclusive access mark is checked by Store Exclusive instructions. See *Synchronization and semaphores*. For information about memory accesses, see *Load/store addressing modes*.



Encoding

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

Decode for this encoding

```
integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tagchecked = n != 31;
```

Assembler symbols

<Wt>	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

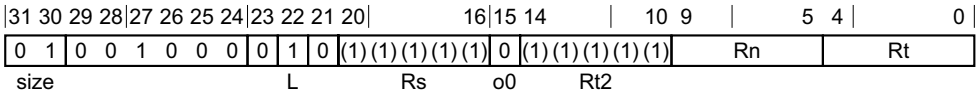
```
bits(64) address;
bits(8) data;

AccessDescriptor accdesc = CreateAccDescExLDST(MemOp_LOAD, FALSE, tagchecked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

// Tell the Exclusives monitors to record a sequence of one or more atomic
// memory reads from virtual address range [address, address+dbytes-1].
// The Exclusives monitor will only be set if all the reads are from the
// same dbytes-aligned physical address, to allow for the possibility of
// an atomicity break if the translation is changed between reads.
AArch64.SetExclusiveMonitors(address, 1);

data = Mem[address, 1, accdesc];
X[t, 32] = ZeroExtend(data, 32);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

C6.2.230 LDXRH

Load Exclusive Register Halfword derives an address from a base register value, loads a halfword from memory, zero-extends it and writes it to a register. The memory access is atomic. The PE marks the physical address being accessed as an exclusive access. This exclusive access mark is checked by Store Exclusive instructions. See *Synchronization and semaphores*. For information about memory accesses, see *Load/store addressing modes*.



Encoding

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

Decode for this encoding

```
integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tagchecked = n != 31;
```

Assembler symbols

<Wt>	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(16) data;

AccessDescriptor accdesc = CreateAccDescExLDST(MemOp_LOAD, FALSE, tagchecked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

// Tell the Exclusives monitors to record a sequence of one or more atomic
// memory reads from virtual address range [address, address+dbytes-1].
// The Exclusives monitor will only be set if all the reads are from the
// same dbytes-aligned physical address, to allow for the possibility of
// an atomicity break if the translation is changed between reads.
AArch64.SetExclusiveMonitors(address, 2);

data = Mem[address, 2, accdesc];
X[t, 32] = ZeroExtend(data, 32);
```

Operational information

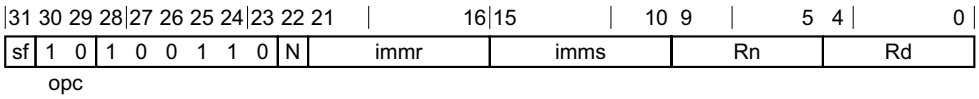
If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

C6.2.231 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 instruction is an alias of the UBFM instruction. This means that:

- The encodings in this description are named to match the encodings of UBFM.
- The description of UBFM gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.



32-bit variant

Applies when `sf == 0 && N == 0 && imms != 011111`.

`LSL <Wd>, <Wn>, #<shift>`

is equivalent to

`UBFM <Wd>, <Wn>, #(-<shift> MOD 32), #(31-<shift>)`

and is the preferred disassembly when `imms + 1 == immr`.

64-bit variant

Applies when `sf == 1 && N == 1 && imms != 111111`.

`LSL <Xd>, <Xn>, #<shift>`

is equivalent to

`UBFM <Xd>, <Xn>, #(-<shift> MOD 64), #(63-<shift>)`

and is the preferred disassembly when `imms + 1 == immr`.

Assembler symbols

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

Operation

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

Operational information

If PSTATE.DIT is 1:

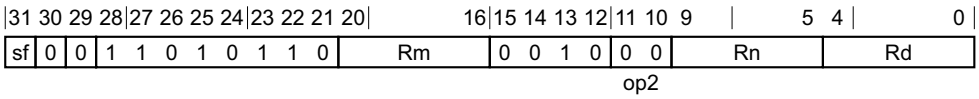
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.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

C6.2.232 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 remainder obtained by dividing the second source register by the data size defines the number of bits by which the first source register is left-shifted.

This instruction is an alias of the LSLV instruction. This means that:

- The encodings in this description are named to match the encodings of LSLV.
- The description of LSLV gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.



32-bit variant

Applies when `sf == 0`.

`LSL <Wd>, <Wn>, <Wm>`

is equivalent to

`LSLV <Wd>, <Wn>, <Wm>`

and is always the preferred disassembly.

64-bit variant

Applies when `sf == 1`.

`LSL <Xd>, <Xn>, <Xm>`

is equivalent to

`LSLV <Xd>, <Xn>, <Xm>`

and is always the preferred disassembly.

Assembler symbols

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

Operation

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

Operational information

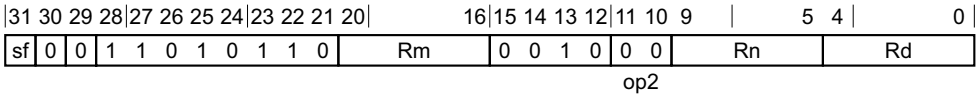
If PSTATE.DIT is 1:

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

C6.2.233 LSLV

Logical Shift Left Variable shifts a register value left by a variable number of bits, shifting in zeros, and writes the result to the destination register. The remainder obtained by dividing the second source register by the data size defines the number of bits by which the first source register is left-shifted.

This instruction is used by the alias LSL (register). The alias is always the preferred disassembly.



32-bit variant

Applies when `sf == 0`.

```
LSLV <Wd>, <Wn>, <Wm>
```

64-bit variant

Applies when `sf == 1`.

```
LSLV <Xd>, <Xn>, <Xm>
```

Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
constant integer datasize = 32 << UInt(sf);
ShiftType shift_type = DecodeShift(op2);
```

Assembler symbols

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

Operation

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

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

Operational information

If PSTATE.DIT is 1:

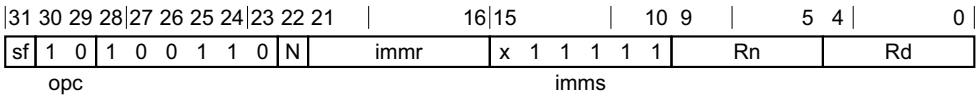
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.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

C6.2.234 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 instruction is an alias of the UBFM instruction. This means that:

- The encodings in this description are named to match the encodings of UBFM.
- The description of UBFM gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.



32-bit variant

Applies when `sf == 0 && N == 0 && imms == 011111`.

`LSR <Wd>, <Wn>, #<shift>`

is equivalent to

`UBFM <Wd>, <Wn>, #<shift>, #31`

and is always the preferred disassembly.

64-bit variant

Applies when `sf == 1 && N == 1 && imms == 111111`.

`LSR <Xd>, <Xn>, #<shift>`

is equivalent to

`UBFM <Xd>, <Xn>, #<shift>, #63`

and is always the preferred disassembly.

Assembler symbols

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

Operation

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

Operational information

If PSTATE.DIT is 1:

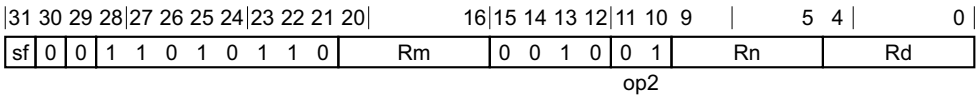
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.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

C6.2.235 LSR (register)

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

This instruction is an alias of the LSRV instruction. This means that:

- The encodings in this description are named to match the encodings of LSRV.
- The description of LSRV gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.



32-bit variant

Applies when `sf == 0`.

`LSR <Wd>, <Wn>, <Wm>`

is equivalent to

`LSRV <Wd>, <Wn>, <Wm>`

and is always the preferred disassembly.

64-bit variant

Applies when `sf == 1`.

`LSR <Xd>, <Xn>, <Xm>`

is equivalent to

`LSRV <Xd>, <Xn>, <Xm>`

and is always the preferred disassembly.

Assembler symbols

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

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

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

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

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

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

Operation

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

Operational information

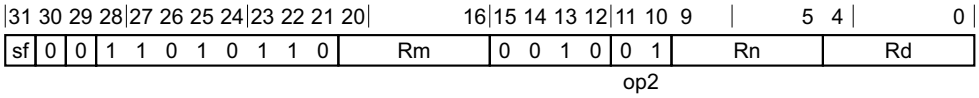
If PSTATE.DIT is 1:

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

C6.2.236 LSRV

Logical Shift Right Variable shifts a register value right by a variable number of bits, shifting in zeros, and writes the result to the destination register. The remainder obtained by dividing the second source register by the data size defines the number of bits by which the first source register is right-shifted.

This instruction is used by the alias LSR (register). The alias is always the preferred disassembly.



32-bit variant

Applies when `sf == 0`.

```
LSRV <Wd>, <Wn>, <Wm>
```

64-bit variant

Applies when `sf == 1`.

```
LSRV <Xd>, <Xn>, <Xm>
```

Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
constant integer datasize = 32 << UInt(sf);
ShiftType shift_type = DecodeShift(op2);
```

Assembler symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Wm>	Is the 32-bit name of the second general-purpose source register holding a shift amount from 0 to 31 in its bottom 5 bits, encoded in the "Rm" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the second general-purpose source register holding a shift amount from 0 to 63 in its bottom 6 bits, encoded in the "Rm" field.

Operation

```
bits(datasize) result;
bits(datasize) operand2 = X[m, datasize];

result = ShiftReg(n, shift_type, UInt(operand2) MOD datasize, datasize);
X[d, datasize] = result;
```

Operational information

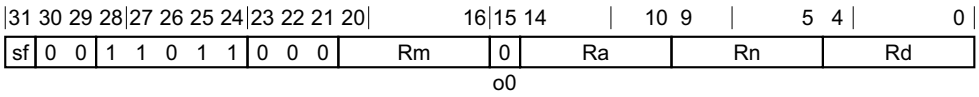
If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

C6.2.237 MADD

Multiply-Add multiplies two register values, adds a third register value, and writes the result to the destination register.

This instruction is used by the alias MUL. See *Alias conditions* for details of when each alias is preferred.



32-bit variant

Applies when `sf == 0`.

```
MADD <Wd>, <Wn>, <Wm>, <Wa>
```

64-bit variant

Applies when `sf == 1`.

```
MADD <Xd>, <Xn>, <Xm>, <Xa>
```

Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer a = UInt(Ra);
constant integer destsize = 32 << UInt(sf);
```

Alias conditions

Alias	is preferred when
MUL	`Ra == '11111'`

Assembler symbols

`<Wd>` Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

`<Wn>` Is the 32-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.

`<Wm>` Is the 32-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.

`<Wa>` Is the 32-bit name of the third general-purpose source register holding the addend, encoded in the "Ra" field.

`<Xd>` Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

`<Xn>` Is the 64-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.

`<Xm>` Is the 64-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.

<Xa> Is the 64-bit name of the third general-purpose source register holding the addend, encoded in the "Ra" field.

Operation

```
bits(destsize) operand1 = X[n, destsize];
bits(destsize) operand2 = X[m, destsize];
bits(destsize) operand3 = X[a, destsize];

integer result;

result = UInt(operand3) + (UInt(operand1) * UInt(operand2));

X[d, destsize] = result<destsize-1:0>;
```

Operational information

If PSTATE.DIT is 1:

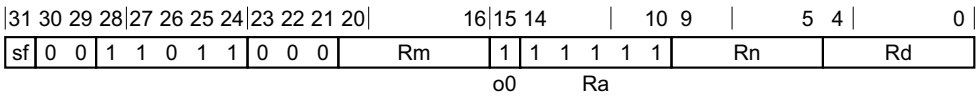
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

C6.2.238 MNEG

Multiply-Negate multiplies two register values, negates the product, and writes the result to the destination register.

This instruction is an alias of the MSUB instruction. This means that:

- The encodings in this description are named to match the encodings of MSUB.
- The description of MSUB gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.



32-bit variant

Applies when `sf == 0`.

```
MNEG <Wd>, <Wn>, <Wm>
```

is equivalent to

```
MSUB <Wd>, <Wn>, <Wm>, WZR
```

and is always the preferred disassembly.

64-bit variant

Applies when `sf == 1`.

```
MNEG <Xd>, <Xn>, <Xm>
```

is equivalent to

```
MSUB <Xd>, <Xn>, <Xm>, XZR
```

and is always the preferred disassembly.

Assembler symbols

`<Wd>`	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
`<Wn>`	Is the 32-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
`<Wm>`	Is the 32-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.
`<Xd>`	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
`<Xn>`	Is the 64-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
`<Xm>`	Is the 64-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.

Operation

The description of MSUB gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

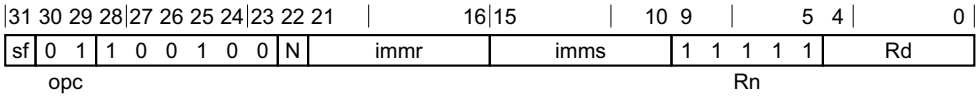
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

C6.2.239 MOV (bitmask immediate)

Move (bitmask immediate) writes a bitmask immediate value to a register.

This instruction is an alias of the ORR (immediate) instruction. This means that:

- The encodings in this description are named to match the encodings of ORR (immediate).
- The description of ORR (immediate) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.



32-bit variant

Applies when `sf == 0 && N == 0`.

```
MOV <Wd|WSP>, #<imm>
```

is equivalent to

```
ORR <Wd|WSP>, WZR, #<imm>
```

and is the preferred disassembly when `! MoveWidePreferred(sf, N, imms, immr)`.

64-bit variant

Applies when `sf == 1`.

```
MOV <Xd|SP>, #<imm>
```

is equivalent to

```
ORR <Xd|SP>, XZR, #<imm>
```

and is the preferred disassembly when `! MoveWidePreferred(sf, N, imms, immr)`.

Assembler symbols

`<Wd|WSP>` Is the 32-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.

`<Xd|SP>` Is the 64-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.

`<imm>` For the 32-bit variant: is the bitmask immediate, encoded in "imms:immr", but excluding values which could be encoded by MOVZ or MOVN.

For the 64-bit variant: is the bitmask immediate, encoded in "N:imms:immr", but excluding values which could be encoded by MOVZ or MOVN.

Operation

The description of ORR (immediate) gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

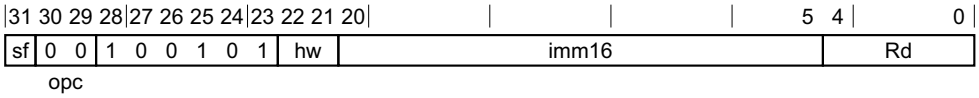
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

C6.2.240 MOV (inverted wide immediate)

Move (inverted wide immediate) moves an inverted 16-bit immediate value to a register.

This instruction is an alias of the MOVN instruction. This means that:

- The encodings in this description are named to match the encodings of MOVN.
- The description of MOVN gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.



32-bit variant

Applies when `sf == 0 && hw == 0x`.

```
MOV <Wd>, #<imm>
```

is equivalent to

```
MOVN <Wd>, #<imm16>, LSL #<shift>
```

and is the preferred disassembly when `! (IsZero(imm16) && hw != '00') && ! IsOnes(imm16)`.

64-bit variant

Applies when `sf == 1`.

```
MOV <Xd>, #<imm>
```

is equivalent to

```
MOVN <Xd>, #<imm16>, LSL #<shift>
```

and is the preferred disassembly when `! (IsZero(imm16) && hw != '00')`.

Assembler symbols

`<Wd>`	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
`<Xd>`	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
`<imm>`	For the 32-bit variant: is a 32-bit immediate, the bitwise inverse of which can be encoded in "imm16:hw", but excluding 0xffff0000 and 0x0000ffff For the 64-bit variant: is a 64-bit immediate, the bitwise inverse of which can be encoded in "imm16:hw".
`<shift>`	For the 32-bit variant: is the amount by which to shift the immediate left, either 0 (the default) or 16, encoded in the "hw" field as <shift>/16. For the 64-bit variant: is the amount by which to shift the immediate left, either 0 (the default), 16, 32 or 48, encoded in the "hw" field as <shift>/16.

Operation

The description of MOVN gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

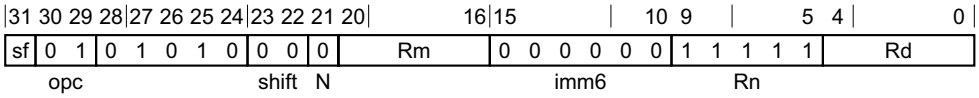
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

C6.2.241 MOV (register)

Move (register) copies the value in a source register to the destination register.

This instruction is an alias of the ORR (shifted register) instruction. This means that:

- The encodings in this description are named to match the encodings of ORR (shifted register).
- The description of ORR (shifted register) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.



32-bit variant

Applies when `sf == 0`.

```
MOV <Wd>, <Wm>
```

is equivalent to

```
ORR <Wd>, WZR, <Wm>
```

and is always the preferred disassembly.

64-bit variant

Applies when `sf == 1`.

```
MOV <Xd>, <Xm>
```

is equivalent to

```
ORR <Xd>, XZR, <Xm>
```

and is always the preferred disassembly.

Assembler symbols

`<Wd>`	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
`<Wm>`	Is the 32-bit name of the general-purpose source register, encoded in the "Rm" field.
`<Xd>`	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
`<Xm>`	Is the 64-bit name of the general-purpose source register, encoded in the "Rm" field.

Operation

The description of ORR (shifted register) gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

C6.2.242 MOV (to/from SP)

Move between register and stack pointer : `Rd = Rn`

This instruction is an alias of the ADD (immediate) instruction. This means that:

- The encodings in this description are named to match the encodings of ADD (immediate).
- The description of ADD (immediate) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

32-bit variant

Applies when `sf == 0`.

`MOV <Wd|WSP>, <Wn|WSP>`

is equivalent to

`ADD <Wd|WSP>, <Wn|WSP>, #0`

and is the preferred disassembly when `(Rd == '11111' || Rn == '11111')`.

64-bit variant

Applies when `sf == 1`.

`MOV <Xd|SP>, <Xn|SP>`

is equivalent to

`ADD <Xd|SP>, <Xn|SP>, #0`

and is the preferred disassembly when `(Rd == '11111' || Rn == '11111')`.

Assembler symbols

`<Wd|WSP>`	Is the 32-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.
`<Wn|WSP>`	Is the 32-bit name of the source general-purpose register or stack pointer, encoded in the "Rn" field.
`<Xd|SP>`	Is the 64-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.
`<Xn|SP>`	Is the 64-bit name of the source general-purpose register or stack pointer, encoded in the "Rn" field.

Operation

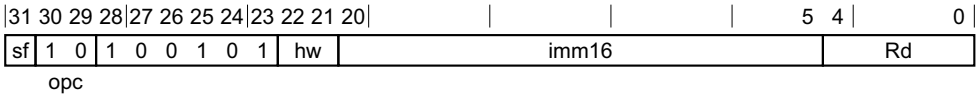
The description of ADD (immediate) gives the operational pseudocode for this instruction.

C6.2.243 MOV (wide immediate)

Move (wide immediate) moves a 16-bit immediate value to a register.

This instruction is an alias of the MOVZ instruction. This means that:

- The encodings in this description are named to match the encodings of MOVZ.
- The description of MOVZ gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.



32-bit variant

Applies when `sf == 0 && hw == 0x`.

```
MOV <Wd>, #<imm>
```

is equivalent to

```
MOVZ <Wd>, #<imm16>, LSL #<shift>
```

and is the preferred disassembly when `! (IsZero(imm16) && hw != '00')`.

64-bit variant

Applies when `sf == 1`.

```
MOV <Xd>, #<imm>
```

is equivalent to

```
MOVZ <Xd>, #<imm16>, LSL #<shift>
```

and is the preferred disassembly when `! (IsZero(imm16) && hw != '00')`.

Assembler symbols

`<Wd>`	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
`<Xd>`	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
`<imm>`	For the 32-bit variant: is a 32-bit immediate which can be encoded in "imm16:hw". For the 64-bit variant: is a 64-bit immediate which can be encoded in "imm16:hw".
`<shift>`	For the 32-bit variant: is the amount by which to shift the immediate left, either 0 (the default) or 16, encoded in the "hw" field as <shift>/16. For the 64-bit variant: is the amount by which to shift the immediate left, either 0 (the default), 16, 32 or 48, encoded in the "hw" field as <shift>/16.

Operation

The description of MOVZ gives the operational pseudocode for this instruction.

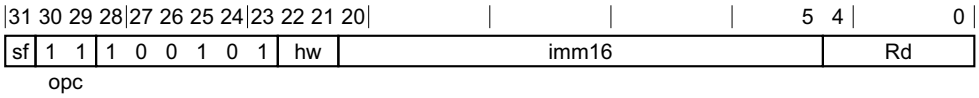
Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

C6.2.244 MOVK

Move wide with keep moves an optionally-shifted 16-bit immediate value into a register, keeping other bits unchanged.



32-bit variant

Applies when `sf == 0 && hw == 0x`.

```
MOVK <Wd>, #<imm>{, LSL #<shift>}
```

64-bit variant

Applies when `sf == 1`.

```
MOVK <Xd>, #<imm>{, LSL #<shift>}
```

Decode for all variants of this encoding

```
if sf == '0' && hw<1> == '1' then UNDEFINED;

integer d = UInt(Rd);
constant integer datasize = 32 << UInt(sf);
constant integer pos = UInt(hw:'0000');
```

Assembler symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<imm>	Is the 16-bit unsigned immediate, in the range 0 to 65535, encoded in the "imm16" field.
<shift>	For the 32-bit variant: is the amount by which to shift the immediate left, either 0 (the default) or 16, encoded in the "hw" field as <shift>/16. For the 64-bit variant: is the amount by which to shift the immediate left, either 0 (the default), 16, 32 or 48, encoded in the "hw" field as <shift>/16.

Operation

```
bits(datasize) result;

result = X[d, datasize];
result<pos+15:pos> = imm16;
X[d, datasize] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.

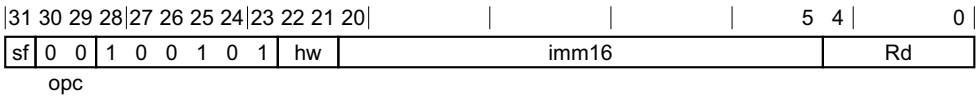
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

C6.2.245 MOVN

Move wide with NOT moves the inverse of an optionally-shifted 16-bit immediate value to a register.

This instruction is used by the alias MOV (inverted wide immediate). See *Alias conditions* for details of when each alias is preferred.



32-bit variant

Applies when sf == 0 && hw == 0x.

```
MOVN <Wd>, #<imm>{, LSL #<shift>}
```

64-bit variant

Applies when sf == 1.

```
MOVN <Xd>, #<imm>{, LSL #<shift>}
```

Decode for all variants of this encoding

```
if sf == '0' && hw<1> == '1' then UNDEFINED;

integer d = UInt(Rd);
constant integer datasize = 32 << UInt(sf);
constant integer pos = UInt(hw:'0000');
```

Alias conditions

Alias	of variant	is preferred when
MOV (inverted wide immediate)	64-bit	! (IsZero(imm16) && hw != '00')
MOV (inverted wide immediate)	32-bit	! (IsZero(imm16) && hw != '00') && ! IsOnes(imm16)

Assembler symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<imm>	Is the 16-bit unsigned immediate, in the range 0 to 65535, encoded in the "imm16" field.
<shift>	For the 32-bit variant: is the amount by which to shift the immediate left, either 0 (the default) or 16, encoded in the "hw" field as <shift>/16. For the 64-bit variant: is the amount by which to shift the immediate left, either 0 (the default), 16, 32 or 48, encoded in the "hw" field as <shift>/16.

Operation

```
bits(datasize) result;

result = Zeros(datasize);
```

```
result<pos+15:pos> = imm16;
result = NOT(result);
X[d, datasize] = result;
```

Operational information

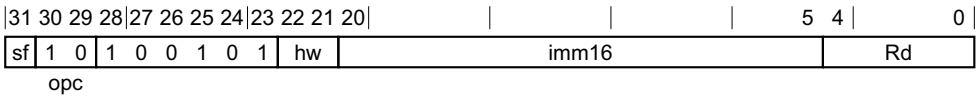
If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

C6.2.246 MOVZ

Move wide with zero moves an optionally-shifted 16-bit immediate value to a register.

This instruction is used by the alias MOV (wide immediate). See *Alias conditions* for details of when each alias is preferred.



32-bit variant

Applies when sf == 0 && hw == 0x.

```
MOVZ <Wd>, #<imm>{, LSL #<shift>}
```

64-bit variant

Applies when sf == 1.

```
MOVZ <Xd>, #<imm>{, LSL #<shift>}
```

Decode for all variants of this encoding

```
if sf == '0' && hw<1> == '1' then UNDEFINED;

integer d = UInt(Rd);
constant integer datasize = 32 << UInt(sf);
constant integer pos = UInt(hw:'0000');
```

Alias conditions

Alias	is preferred when
MOV (wide immediate)	! (IsZero(imm16) && hw != '00')

Assembler symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<imm>	Is the 16-bit unsigned immediate, in the range 0 to 65535, encoded in the "imm16" field.
<shift>	For the 32-bit variant: is the amount by which to shift the immediate left, either 0 (the default) or 16, encoded in the "hw" field as <shift>/16. For the 64-bit variant: is the amount by which to shift the immediate left, either 0 (the default), 16, 32 or 48, encoded in the "hw" field as <shift>/16.

Operation

```
bits(datasize) result;

result = Zeros(datasize);
```

```
result<pos+15:pos> = imm16;
X[d, datasize] = result;
```

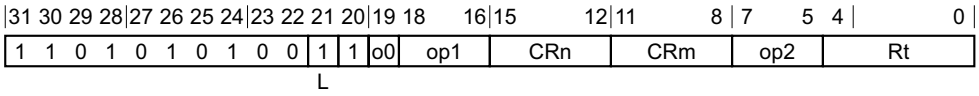
Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

C6.2.247 MRS

Move System Register to general-purpose register allows the PE to read an AArch64 System register into a general-purpose register.



Encoding

```
MRS <Xt>, (<systemreg>|S<op0>_<op1>_<Cn>_<Cm>_<op2>)
```

Decode for this encoding

```
AArch64.CheckSystemAccess('1':o0, op1, CRn, CRm, op2, Rt, L);

integer t = UInt(Rt);

integer sys_op0 = 2 + UInt(o0);
integer sys_op1 = UInt(op1);
integer sys_op2 = UInt(op2);
integer sys_crn = UInt(CRn);
integer sys_crm = UInt(CRm);
```

Assembler symbols

<Xt>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rt" field.
<systemreg>	Is a System register name, encoded in the "o0:op1:CRn:CRm:op2". The System register names are defined in Chapter D23 *AArch64 System Register Descriptions*.
<op0>	Is an unsigned immediate, encoded in the "o0" field. It can have the following values: 2 when o0 = 0 3 when o0 = 1
<op1>	Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op1" field.
<Cn>	Is a name 'Cn', with 'n' in the range 0 to 15, encoded in the "CRn" field.
<Cm>	Is a name 'Cm', with 'm' in the range 0 to 15, encoded in the "CRm" field.
<op2>	Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op2" field.

Operation

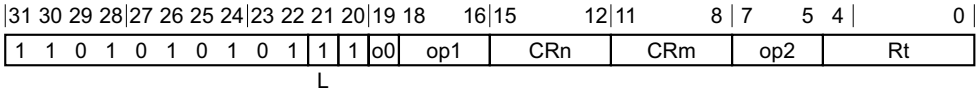
```
AArch64.SysRegRead(sys_op0, sys_op1, sys_crn, sys_crm, sys_op2, t);
```

C6.2.248 MRRS

Move System Register to two adjacent general-purpose registers allows the PE to read an AArch64 128-bit System register into two adjacent 64-bit general-purpose registers.

System

(FEAT_SYSREG128)



Encoding

```
MRRS <Xt>, <Xt+1>, (<systemreg>|S<op0>_<op1>_<Cn>_<Cm>_<op2>)
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SYSREG128) then UNDEFINED;
if Rt<0> == '1' then UNDEFINED;
AArch64.CheckSystemAccess('1':o0, op1, CRn, CRm, op2, Rt, L);

integer t = UInt(Rt);
integer t2 = UInt(Rt + 1);

integer sys_op0 = 2 + UInt(o0);
integer sys_op1 = UInt(op1);
integer sys_op2 = UInt(op2);
integer sys_crn = UInt(CRn);
integer sys_crm = UInt(CRm);
```

Assembler symbols

<Xt>	Is the 64-bit name of the first general-purpose destination register, encoded in the "Rt" field.
<Xt+1>	Is the 64-bit name of the second general-purpose destination register, encoded as "Rt" +1.
<systemreg>	Is a System register name, encoded in "o0:op1:CRn:CRm:op2".
<op0>	Is an unsigned immediate, encoded in the "o0" field. It can have the following values: 2 when o0 = 0 3 when o0 = 1
<op1>	Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op1" field.
<Cn>	Is a name 'Cn', with 'n' in the range 0 to 15, encoded in the "CRn" field.
<Cm>	Is a name 'Cm', with 'm' in the range 0 to 15, encoded in the "CRm" field.
<op2>	Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op2" field.

Operation

```
AArch64.SysRegRead128(sys_op0, sys_op1, sys_crn, sys_crm, sys_op2, t, t2);
```

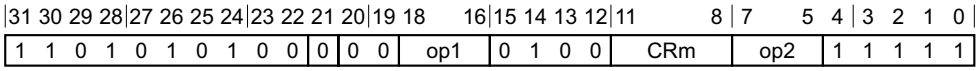
C6.2.249 MSR (immediate)

Move immediate value to Special Register moves an immediate value to selected bits of the PSTATE. For more information, see PSTATE.

The bits that can be written by this instruction are:

- PSTATE.D, PSTATE.A, PSTATE.I, PSTATE.F, and PSTATE.SP.
- If FEAT_SSBS is implemented, PSTATE.SSBS.
- If FEAT_PAN is implemented, PSTATE.PAN.
- If FEAT_UAO is implemented, PSTATE.UAO.
- If FEAT_DIT is implemented, PSTATE.DIT.
- If FEAT_MTE is implemented, PSTATE.TCO.
- If FEAT_NMI is implemented, PSTATE.ALLINT.
- If FEAT_SME is implemented, PSTATE.SM and PSTATE.ZA.
- If FEAT_EBEP is implemented, PSTATE.PM.

This instruction is used by the aliases SMSTART and SMSTOP. See *Alias conditions* for details of when each alias is preferred.



Encoding

```
MSR <pstatefield>, #<imm>
```

Decode for this encoding

```
if op1 == '000' && op2 == '000' then SEE "CFINV";
if op1 == '000' && op2 == '001' then SEE "XAFLAG";
if op1 == '000' && op2 == '010' then SEE "AXFLAG";

AArch64.CheckSystemAccess('00', op1, '0100', CRm, op2, '11111', '0');
bits(2) min_EL;
boolean need_secure = FALSE;

case op1 of
    when '00x'
        min_EL = EL1;
    when '010'
        min_EL = EL1;
    when '011'
        min_EL = EL0;
    when '100'
        min_EL = EL2;
    when '101'
        if !IsFeatureImplemented(FEAT_VHE) then
            UNDEFINED;
        min_EL = EL2;
    when '110'
        min_EL = EL3;
    when '111'
        min_EL = EL1;
```

```
        need_secure = TRUE;

if (UInt(PSTATE.EL) < UInt(min_EL) || (need_secure && CurrentSecurityState() != SS_Secure)) then
    UNDEFINED;

PSTATEField field;
case op1:op2 of
    when '000 011'
        if !IsFeatureImplemented(FEAT_UAO) then UNDEFINED;
        field = PSTATEField_UAO;
    when '000 100'
        if !IsFeatureImplemented(FEAT_PAN) then UNDEFINED;
        field = PSTATEField_PAN;
    when '000 101' field = PSTATEField_SP;
    when '001 000'
        case CRm of
            when '000x'
                if !IsFeatureImplemented(FEAT_NMI) then UNDEFINED;
                field = PSTATEField_ALLINT;
            when '001x'
                if !IsFeatureImplemented(FEAT_EBEP) then UNDEFINED;
                field = PSTATEField_PM;
            otherwise
                UNDEFINED;
    when '011 010'
        if !IsFeatureImplemented(FEAT_DIT) then UNDEFINED;
        field = PSTATEField_DIT;
    when '011 011'
        case CRm of
            when '001x'
                if !IsFeatureImplemented(FEAT_SME) then UNDEFINED;
                field = PSTATEField_SVCRSM;
            when '010x'
                if !IsFeatureImplemented(FEAT_SME) then UNDEFINED;
                field = PSTATEField_SVCRZA;
            when '011x'
                if !IsFeatureImplemented(FEAT_SME) then UNDEFINED;
                field = PSTATEField_SVCRSMZA;
            otherwise
                UNDEFINED;
    when '011 100'
        if !IsFeatureImplemented(FEAT_MTE) then UNDEFINED;
        field = PSTATEField_TCO;
    when '011 110' field = PSTATEField_DAIFSet;
    when '011 111' field = PSTATEField_DAIFClr;
    when '011 001'
        if !IsFeatureImplemented(FEAT_SSBS) then UNDEFINED;
        field = PSTATEField_SSBS;
    otherwise UNDEFINED;
```

Alias conditions

Alias	is preferred when
SMSTART	`op1 == '011' && CRm == '0xx1' && op2 == '011'`
SMSTOP	`op1 == '011' && CRm == '0xx0' && op2 == '011'`

Assembler symbols

<pstatefield> Is a PSTATE field name. For the MSR instruction, this is encoded in the "op1:op2:CRm" field. It can have the following values:

SPSel when `op1 = 000`, `op2 = 101`, `CRm = xxxx`

DAIFSet when op1 = 011, op2 = 110, CRm = xxxx

DAIFClr when op1 = 011, op2 = 111, CRm = xxxx

When FEAT_UAO is implemented, the following value is also valid:

UAO when op1 = 000, op2 = 011, CRm = xxxx

When FEAT_PAN is implemented, the following value is also valid:

PAN when op1 = 000, op2 = 100, CRm = xxxx

When FEAT_NMI is implemented, the following value is also valid:

ALLINT when op1 = 001, op2 = 000, CRm = 000x

When FEAT_EBEP is implemented, the following value is also valid:

PM when op1 = 001, op2 = 000, CRm = 001x

When FEAT_SSBS is implemented, the following value is also valid:

SSBS when op1 = 011, op2 = 001, CRm = xxxx

When FEAT_DIT is implemented, the following value is also valid:

DIT when op1 = 011, op2 = 010, CRm = xxxx

When FEAT_SME is implemented, the following values are also valid:

SVCRSM when op1 = 011, op2 = 011, CRm = 001x

SVCRZA when op1 = 011, op2 = 011, CRm = 010x

SVCRSMZA when op1 = 011, op2 = 011, CRm = 011x

When FEAT_MTE is implemented, the following value is also valid:

TCO when op1 = 011, op2 = 100, CRm = xxxx

See *PSTATE* when op1 = 000, op2 = 00x, CRm = xxxx.

See *PSTATE* when op1 = 000, op2 = 010, CRm = xxxx.

The following encodings are reserved:

- op1 = 000, op2 = 11x, CRm = xxxx.
- op1 = 001, op2 = 000, CRm = 01xx.
- op1 = 001, op2 = 000, CRm = 1xxx.
- op1 = 001, op2 = 001, CRm = xxxx.
- op1 = 001, op2 = 01x, CRm = xxxx.
- op1 = 001, op2 = 1xx, CRm = xxxx.
- op1 = 010, op2 = xxx, CRm = xxxx.
- op1 = 011, op2 = 000, CRm = xxxx.
- op1 = 011, op2 = 011, CRm = 000x.
- op1 = 011, op2 = 011, CRm = 1xxx.
- op1 = 011, op2 = 101, CRm = xxxx.
- op1 = 1xx, op2 = xxx, CRm = xxxx.

For the SMSTART and SMSTOP aliases, this is encoded in "CRm<2:1>", where 0b01 specifies SVCRSM, 0b10 specifies SVCRZA, and 0b11 specifies SVCRSMZA.

<imm> Is a 4-bit unsigned immediate, in the range 0 to 15, encoded in the "CRm" field. Restricted to the range 0 to 1, encoded in "CRm<0>", when <pstatefield> is ALLINT, PM, SVCRSM, SVCRSMZA, or SVCRZA.

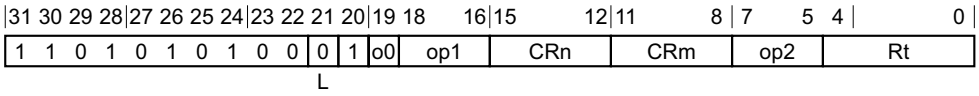
Operation

```
case field of
    when PSTATEField_SSBS
        PSTATE.SSBS = CRm<0>;
```

```
when PSTATEField_SP
    PSTATE.SP = CRm<0>;
when PSTATEField_DAIFSet
    AArch64.CheckDAIFAccess(PSTATEField_DAIFSet);
    PSTATE.D = PSTATE.D OR CRm<3>;
    PSTATE.A = PSTATE.A OR CRm<2>;
    PSTATE.I = PSTATE.I OR CRm<1>;
    PSTATE.F = PSTATE.F OR CRm<0>;
when PSTATEField_DAIFClr
    AArch64.CheckDAIFAccess(PSTATEField_DAIFClr);
    PSTATE.D = PSTATE.D AND NOT(CRm<3>);
    PSTATE.A = PSTATE.A AND NOT(CRm<2>);
    PSTATE.I = PSTATE.I AND NOT(CRm<1>);
    PSTATE.F = PSTATE.F AND NOT(CRm<0>);
when PSTATEField_PAN
    PSTATE.PAN = CRm<0>;
when PSTATEField_UAO
    PSTATE.UAO = CRm<0>;
when PSTATEField_DIT
    PSTATE.DIT = CRm<0>;
when PSTATEField_TCO
    PSTATE.TCO = CRm<0>;
when PSTATEField_ALLINT
    if (PSTATE.EL == EL1 && IsHCRXEL2Enabled() && HCRX_EL2.TALLINT == '1' && CRm<0> == '1') then
        AArch64.SystemAccessTrap(EL2, 0x18);
    PSTATE.ALLINT = CRm<0>;
when PSTATEField_SVCRSM
    CheckSMEAccess();
    SetPSTATE_SM(CRm<0>);
when PSTATEField_SVCRZA
    CheckSMEAccess();
    SetPSTATE_ZA(CRm<0>);
when PSTATEField_SVCRSMZA
    CheckSMEAccess();
    SetPSTATE_SM(CRm<0>);
    SetPSTATE_ZA(CRm<0>);
when PSTATEField_PM
    PSTATE.PM = CRm<0>;
```

C6.2.250 MSR (register)

Move general-purpose register to System Register allows the PE to write an AArch64 System register from a general-purpose register.



Encoding

```
MSR (<systemreg>|S<op0>_<op1>_<Cn>_<Cm>_<op2>), <Xt>
```

Decode for this encoding

```
AArch64.CheckSystemAccess('1':o0, op1, CRn, CRm, op2, Rt, L);

integer t = UInt(Rt);

integer sys_op0 = 2 + UInt(o0);
integer sys_op1 = UInt(op1);
integer sys_op2 = UInt(op2);
integer sys_crn = UInt(CRn);
integer sys_crm = UInt(CRm);
```

Assembler symbols

<systemreg>	Is a System register name, encoded in the "o0:op1:CRn:CRm:op2". The System register names are defined in Chapter D23 *AArch64 System Register Descriptions*.
<op0>	Is an unsigned immediate, encoded in the "o0" field. It can have the following values: 2 when o0 = 0 3 when o0 = 1
<op1>	Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op1" field.
<Cn>	Is a name 'Cn', with 'n' in the range 0 to 15, encoded in the "CRn" field.
<Cm>	Is a name 'Cm', with 'm' in the range 0 to 15, encoded in the "CRm" field.
<op2>	Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op2" field.
<Xt>	Is the 64-bit name of the general-purpose source register, encoded in the "Rt" field.

Operation

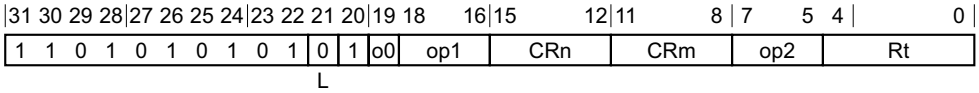
```
AArch64.SysRegWrite(sys_op0, sys_op1, sys_crn, sys_crm, sys_op2, t);
```

C6.2.251 MSRR

Move two adjacent general-purpose registers to System Register allows the PE to write an AArch64 128-bit System register from two adjacent 64-bit general-purpose registers.

System

(FEAT_SYSREG128)



Encoding

```
MSRR (<systemreg>|S<op0>_<op1>_<Cn>_<Cm>_<op2>), <Xt>, <Xt+1>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SYSREG128) then UNDEFINED;
if Rt<0> == '1' then UNDEFINED;
AArch64.CheckSystemAccess('1':o0, op1, CRn, CRm, op2, Rt, L);

integer t = UInt(Rt);
integer t2 = UInt(Rt + 1);

integer sys_op0 = 2 + UInt(o0);
integer sys_op1 = UInt(op1);
integer sys_op2 = UInt(op2);
integer sys_crn = UInt(CRn);
integer sys_crm = UInt(CRm);
```

Assembler symbols

<systemreg>	Is a System register name, encoded in "o0:op1:CRn:CRm:op2".
<op0>	Is an unsigned immediate, encoded in the "o0" field. It can have the following values: 2 when o0 = 0 3 when o0 = 1
<op1>	Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op1" field.
<Cn>	Is a name 'Cn', with 'n' in the range 0 to 15, encoded in the "CRn" field.
<Cm>	Is a name 'Cm', with 'm' in the range 0 to 15, encoded in the "CRm" field.
<op2>	Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op2" field.
<Xt>	Is the 64-bit name of the first general-purpose source register, encoded in the "Rt" field.
<Xt+1>	Is the 64-bit name of the second general-purpose source register, encoded as "Rt" +1.

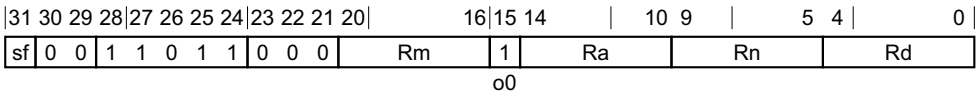
Operation

```
AArch64.SysRegWrite128(sys_op0, sys_op1, sys_crn, sys_crm, sys_op2, t, t2);
```

C6.2.252 MSUB

Multiply-Subtract multiplies two register values, subtracts the product from a third register value, and writes the result to the destination register.

This instruction is used by the alias MNEG. See *Alias conditions* for details of when each alias is preferred.



32-bit variant

Applies when `sf == 0`.

```
MSUB <Wd>, <Wn>, <Wm>, <Wa>
```

64-bit variant

Applies when `sf == 1`.

```
MSUB <Xd>, <Xn>, <Xm>, <Xa>
```

Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer a = UInt(Ra);
constant integer destsize = 32 << UInt(sf);
```

Alias conditions

Alias	is preferred when
MNEG	`Ra == '11111'`

Assembler symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
<Wm>	Is the 32-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.
<Wa>	Is the 32-bit name of the third general-purpose source register holding the minuend, encoded in the "Ra" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.

<Xa> Is the 64-bit name of the third general-purpose source register holding the minuend, encoded in the "Ra" field.

Operation

```
bits(destsize) operand1 = X[n, destsize];
bits(destsize) operand2 = X[m, destsize];
bits(destsize) operand3 = X[a, destsize];

integer result;

result = UInt(operand3) - (UInt(operand1) * UInt(operand2));
X[d, destsize] = result<destsize-1:0>;
```

Operational information

If PSTATE.DIT is 1:

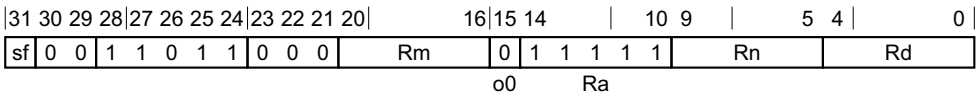
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

C6.2.253 MUL

Multiply : `Rd = Rn * Rm`

This instruction is an alias of the MADD instruction. This means that:

- The encodings in this description are named to match the encodings of MADD.
- The description of MADD gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.



32-bit variant

Applies when `sf == 0`.

`MUL <Wd>, <Wn>, <Wm>`

is equivalent to

`MADD <Wd>, <Wn>, <Wm>, WZR`

and is always the preferred disassembly.

64-bit variant

Applies when `sf == 1`.

`MUL <Xd>, <Xn>, <Xm>`

is equivalent to

`MADD <Xd>, <Xn>, <Xm>, XZR`

and is always the preferred disassembly.

Assembler symbols

`<Wd>` Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

`<Wn>` Is the 32-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.

`<Wm>` Is the 32-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.

`<Xd>` Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

`<Xn>` Is the 64-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.

`<Xm>` Is the 64-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.

Operation

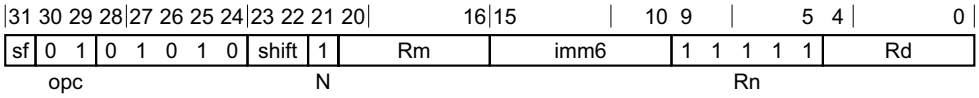
The description of MADD gives the operational pseudocode for this instruction.

C6.2.254 MVN

Bitwise NOT writes the bitwise inverse of a register value to the destination register.

This instruction is an alias of the ORN (shifted register) instruction. This means that:

- The encodings in this description are named to match the encodings of ORN (shifted register).
- The description of ORN (shifted register) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.



32-bit variant

Applies when `sf == 0`.

```
MVN <Wd>, <Wm>{, <shift> #<amount>}
```

is equivalent to

```
ORN <Wd>, WZR, <Wm>{, <shift> #<amount>}
```

and is always the preferred disassembly.

64-bit variant

Applies when `sf == 1`.

```
MVN <Xd>, <Xm>{, <shift> #<amount>}
```

is equivalent to

```
ORN <Xd>, XZR, <Xm>{, <shift> #<amount>}
```

and is always the preferred disassembly.

Assembler symbols

`<Wd>` Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

`<Wm>` Is the 32-bit name of the general-purpose source register, encoded in the "Rm" field.

`<Xd>` Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

`<Xm>` Is the 64-bit name of the general-purpose source register, encoded in the "Rm" field.

`<shift>` Is the optional shift to be applied to the final source, defaulting to LSL and encoded in the "shift" field. It can have the following values:

- `LSL` when `shift = 00`
- `LSR` when `shift = 01`
- `ASR` when `shift = 10`
- `ROR` when `shift = 11`

`<amount>` For the 32-bit variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field.

For the 64-bit variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field,

Operation

The description of ORN (shifted register) gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

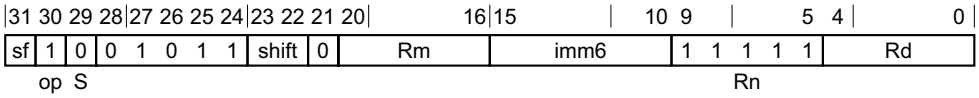
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

C6.2.255 NEG (shifted register)

Negate (shifted register) negates an optionally-shifted register value, and writes the result to the destination register.

This instruction is an alias of the SUB (shifted register) instruction. This means that:

- The encodings in this description are named to match the encodings of SUB (shifted register).
- The description of SUB (shifted register) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.



32-bit variant

Applies when `sf == 0`.

```
NEG <Wd>, <Wm>{, <shift> #<amount>}
```

is equivalent to

```
SUB <Wd>, WZR, <Wm> {, <shift> #<amount>}
```

and is always the preferred disassembly.

64-bit variant

Applies when `sf == 1`.

```
NEG <Xd>, <Xm>{, <shift> #<amount>}
```

is equivalent to

```
SUB <Xd>, XZR, <Xm> {, <shift> #<amount>}
```

and is always the preferred disassembly.

Assembler symbols

`<Wd>` Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

`<Wm>` Is the 32-bit name of the general-purpose source register, encoded in the "Rm" field.

`<Xd>` Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

`<Xm>` Is the 64-bit name of the general-purpose source register, encoded in the "Rm" field.

`<shift>` Is the optional shift type to be applied to the second source operand, defaulting to LSL and encoded in the "shift" field. It can have the following values:

- `LSL` when `shift = 00`
- `LSR` when `shift = 01`
- `ASR` when `shift = 10`

The encoding `shift = 11` is reserved.

`<amount>` For the 32-bit variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field.

For the 64-bit variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field.

Operation

The description of SUB (shifted register) gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

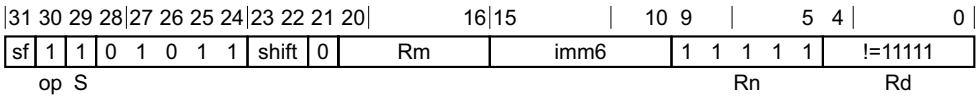
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

C6.2.256 NEGS

Negate, setting flags, negates an optionally-shifted register value, and writes the result to the destination register. It updates the condition flags based on the result.

This instruction is an alias of the SUBS (shifted register) instruction. This means that:

- The encodings in this description are named to match the encodings of SUBS (shifted register).
- The description of SUBS (shifted register) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.



32-bit variant

Applies when `sf == 0`.

```
NEGS <Wd>, <Wm>{, <shift> #<amount>}
```

is equivalent to

```
SUBS <Wd>, WZR, <Wm> {, <shift> #<amount>}
```

and is always the preferred disassembly.

64-bit variant

Applies when `sf == 1`.

```
NEGS <Xd>, <Xm>{, <shift> #<amount>}
```

is equivalent to

```
SUBS <Xd>, XZR, <Xm> {, <shift> #<amount>}
```

and is always the preferred disassembly.

Assembler symbols

`<Wd>` Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

`<Wm>` Is the 32-bit name of the general-purpose source register, encoded in the "Rm" field.

`<Xd>` Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

`<Xm>` Is the 64-bit name of the general-purpose source register, encoded in the "Rm" field.

`<shift>` Is the optional shift type to be applied to the second source operand, defaulting to LSL and encoded in the "shift" field. It can have the following values:

- `LSL` when `shift = 00`
- `LSR` when `shift = 01`
- `ASR` when `shift = 10`

The encoding `shift = 11` is reserved.

`<amount>` For the 32-bit variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field.

For the 64-bit variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field.

Operation

The description of SUBS (shifted register) gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

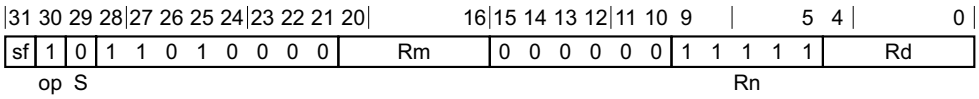
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

C6.2.257 NGC

Negate with Carry negates the sum of a register value and the value of NOT (Carry flag), and writes the result to the destination register.

This instruction is an alias of the SBC instruction. This means that:

- The encodings in this description are named to match the encodings of SBC.
- The description of SBC gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.



32-bit variant

Applies when `sf == 0`.

`NGC <Wd>, <Wm>`

is equivalent to

`SBC <Wd>, WZR, <Wm>`

and is always the preferred disassembly.

64-bit variant

Applies when `sf == 1`.

`NGC <Xd>, <Xm>`

is equivalent to

`SBC <Xd>, XZR, <Xm>`

and is always the preferred disassembly.

Assembler symbols

`<Wd>`	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
`<Wm>`	Is the 32-bit name of the general-purpose source register, encoded in the "Rm" field.
`<Xd>`	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
`<Xm>`	Is the 64-bit name of the general-purpose source register, encoded in the "Rm" field.

Operation

The description of SBC gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.

  - The values of the NZCV flags.

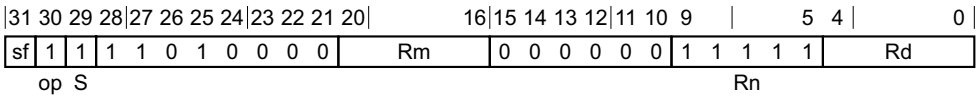
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

C6.2.258 NGCS

Negate with Carry, setting flags, negates the sum of a register value and the value of NOT (Carry flag), and writes the result to the destination register. It updates the condition flags based on the result.

This instruction is an alias of the SBCS instruction. This means that:

- The encodings in this description are named to match the encodings of SBCS.
- The description of SBCS gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.



32-bit variant

Applies when `sf == 0`.

`NGCS <Wd>, <Wm>`

is equivalent to

`SBCS <Wd>, WZR, <Wm>`

and is always the preferred disassembly.

64-bit variant

Applies when `sf == 1`.

`NGCS <Xd>, <Xm>`

is equivalent to

`SBCS <Xd>, XZR, <Xm>`

and is always the preferred disassembly.

Assembler symbols

`<Wd>`	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
`<Wm>`	Is the 32-bit name of the general-purpose source register, encoded in the "Rm" field.
`<Xd>`	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
`<Xm>`	Is the 64-bit name of the general-purpose source register, encoded in the "Rm" field.

Operation

The description of SBCS gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.

  - The values of the NZCV flags.

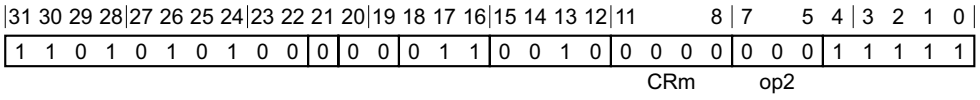
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

C6.2.259 NOP

No Operation does nothing, other than advance the value of the program counter by 4. This instruction can be used for instruction alignment purposes.

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.



Encoding

`NOP`

Decode for this encoding

```
// Empty.
```

Operation

```
return;    // do nothing
```

Operational information

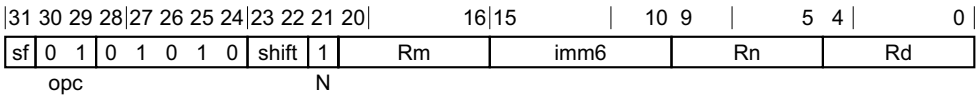
If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

C6.2.260 ORN (shifted register)

Bitwise OR NOT (shifted register) performs a bitwise (inclusive) OR of a register value and the complement of an optionally-shifted register value, and writes the result to the destination register.

This instruction is used by the alias MVN. See *Alias conditions* for details of when each alias is preferred.



32-bit variant

Applies when `sf == 0`.

```
ORN <Wd>, <Wn>, <Wm>{, <shift> #<amount>}
```

64-bit variant

Applies when `sf == 1`.

```
ORN <Xd>, <Xn>, <Xm>{, <shift> #<amount>}
```

Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
constant integer datasize = 32 << UInt(sf);
if sf == '0' && imm6<5> == '1' then UNDEFINED;

ShiftType shift_type = DecodeShift(shift);
integer shift_amount = UInt(imm6);
```

Alias conditions

Alias	is preferred when
MVN	`Rn == '11111'`

Assembler symbols

`<Wd>` Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

`<Wn>` Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.

`<Wm>` Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.

`<Xd>` Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

`<Xn>` Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.

`<Xm>` Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.

`<shift>` Is the optional shift to be applied to the final source, defaulting to LSL and encoded in the "shift" field. It can have the following values:

`LSL` when `shift = 00`

LSR when shift = 01

ASR when shift = 10

ROR when shift = 11

<amount> For the 32-bit variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field.

For the 64-bit variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field,

Operation

```
bits(datasize) operand1 = X[n, datasize];
bits(datasize) operand2 = ShiftReg(m, shift_type, shift_amount, datasize);
bits(datasize) result;

operand2 = NOT(operand2);

result = operand1 OR operand2;
X[d, datasize] = result;
```

Operational information

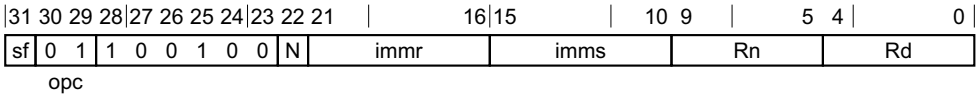
If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

C6.2.261 ORR (immediate)

Bitwise OR (immediate) performs a bitwise (inclusive) OR of a register value and an immediate register value, and writes the result to the destination register.

This instruction is used by the alias MOV (bitmask immediate). See *Alias conditions* for details of when each alias is preferred.



32-bit variant

Applies when `sf == 0 && N == 0`.

```
ORR <Wd|WSP>, <Wn>, #<imm>
```

64-bit variant

Applies when `sf == 1`.

```
ORR <Xd|SP>, <Xn>, #<imm>
```

Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
constant integer datasize = 32 << UInt(sf);
bits(datasize) imm;
if sf == '0' && N != '0' then UNDEFINED;
(imm, -) = DecodeBitMasks(N, imms, immr, TRUE, datasize);
```

Alias conditions

Alias	is preferred when
MOV (bitmask immediate)	`Rn == '11111' && ! MoveWidePreferred(sf, N, imms, immr)`

Assembler symbols

`<Wd|WSP>` Is the 32-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.

`<Wn>` Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.

`<Xd|SP>` Is the 64-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.

`<Xn>` Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.

`<imm>` For the 32-bit variant: is the bitmask immediate, encoded in "imms:immr".

For the 64-bit variant: is the bitmask immediate, encoded in "N:imms:immr".

Operation

```
bits(datasize) result;
bits(datasize) operand1 = X[n, datasize];

result = operand1 OR imm;
if d == 31 then
    SP[] = ZeroExtend(result, 64);
else
    X[d, datasize] = result;
```

Operational information

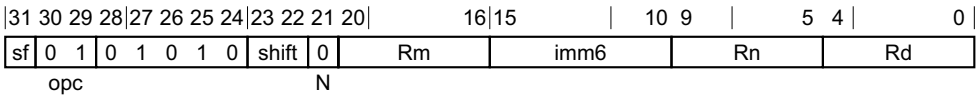
If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

C6.2.262 ORR (shifted register)

Bitwise OR (shifted register) performs a bitwise (inclusive) OR of a register value and an optionally-shifted register value, and writes the result to the destination register.

This instruction is used by the alias MOV (register). See *Alias conditions* for details of when each alias is preferred.



32-bit variant

Applies when `sf == 0`.

```
ORR <Wd>, <Wn>, <Wm>{, <shift> #<amount>}
```

64-bit variant

Applies when `sf == 1`.

```
ORR <Xd>, <Xn>, <Xm>{, <shift> #<amount>}
```

Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
constant integer datasize = 32 << UInt(sf);
if sf == '0' && imm6<5> == '1' then UNDEFINED;

ShiftType shift_type = DecodeShift(shift);
integer shift_amount = UInt(imm6);
```

Alias conditions

Alias	is preferred when
MOV (register)	`shift == '00' && imm6 == '000000' && Rn == '11111'`

Assembler symbols

`<Wd>` Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

`<Wn>` Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.

`<Wm>` Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.

`<Xd>` Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

`<Xn>` Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.

`<Xm>` Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.

`<shift>` Is the optional shift to be applied to the final source, defaulting to LSL and encoded in the "shift" field. It can have the following values:

`LSL` when `shift = 00`

LSR when shift = 01

ASR when shift = 10

ROR when shift = 11

<amount> For the 32-bit variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field.

For the 64-bit variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field,

Operation

```
bits(datasize) operand1 = X[n, datasize];
bits(datasize) operand2 = ShiftReg(m, shift_type, shift_amount, datasize);
bits(datasize) result;

result = operand1 OR operand2;
X[d, datasize] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

C6.2.263 PACDA, PACDZA

Pointer Authentication Code for Data address, using key A. This instruction computes and inserts a pointer authentication code for a data address, using a modifier and key A.

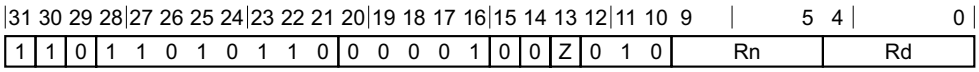
The address is in the general-purpose register that is specified by <Xd>.

The modifier is:

- In the general-purpose register or stack pointer that is specified by <Xn|SP> for PACDA.
- The value zero, for PACDZA.

Integer

(FEAT_PAuth)



PACDA variant

Applies when Z == 0.

```
PACDA <Xd>, <Xn|SP>
```

PACDZA variant

Applies when Z == 1 && Rn == 11111.

```
PACDZA <Xd>
```

Decode for all variants of this encoding

```
boolean source_is_sp = FALSE;
integer d = UInt(Rd);
integer n = UInt(Rn);

if !IsFeatureImplemented(FEAT_PAuth) then
    UNDEFINED;

if Z == '0' then // PACDA
    if n == 31 then source_is_sp = TRUE;
else // PACDZA
    if n != 31 then UNDEFINED;
```

Assembler symbols

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Xn|SP> Is the 64-bit name of the general-purpose source register or stack pointer, encoded in the "Rn" field.

Operation

```
if source_is_sp then
    X[d, 64] = AddPACDA(X[d, 64], SP[]);
else
    X[d, 64] = AddPACDA(X[d, 64], X[n, 64]);
```

C6.2.264 PACDB, PACDZB

Pointer Authentication Code for Data address, using key B. This instruction computes and inserts a pointer authentication code for a data address, using a modifier and key B.

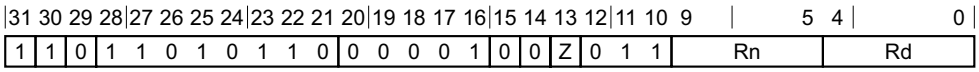
The address is in the general-purpose register that is specified by <Xd>.

The modifier is:

- In the general-purpose register or stack pointer that is specified by <Xn|SP> for PACDB.
- The value zero, for PACDZB.

Integer

(FEAT_PAuth)



PACDB variant

Applies when Z == 0.

```
PACDB <Xd>, <Xn|SP>
```

PACDZB variant

Applies when Z == 1 && Rn == 11111.

```
PACDZB <Xd>
```

Decode for all variants of this encoding

```
boolean source_is_sp = FALSE;
integer d = UInt(Rd);
integer n = UInt(Rn);

if !IsFeatureImplemented(FEAT_PAuth) then
    UNDEFINED;

if Z == '0' then // PACDB
    if n == 31 then source_is_sp = TRUE;
else // PACDZB
    if n != 31 then UNDEFINED;
```

Assembler symbols

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Xn|SP> Is the 64-bit name of the general-purpose source register or stack pointer, encoded in the "Rn" field.

Operation

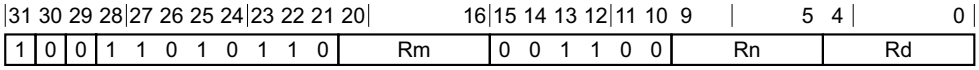
```
if source_is_sp then
    X[d, 64] = AddPACDB(X[d, 64], SP[]);
else
    X[d, 64] = AddPACDB(X[d, 64], X[n, 64]);
```

C6.2.265 PACGA

Pointer Authentication Code, using Generic key. This instruction computes the pointer authentication code for a 64-bit value in the first source register, using a modifier in the second source register, and the Generic key. The computed pointer authentication code is written to the most significant 32 bits of the destination register, and the least significant 32 bits of the destination register are set to zero.

Integer

(FEAT_PAuth)



Encoding

```
PACGA <Xd>, <Xn>, <Xm|SP>
```

Decode for this encoding

```
boolean source_is_sp = FALSE;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if !IsFeatureImplemented(FEAT_PAuth) then
    UNDEFINED;

if m == 31 then source_is_sp = TRUE;
```

Assembler symbols

<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Xm|SP>	Is the 64-bit name of the second general-purpose source register or stack pointer, encoded in the "Rm" field.

Operation

```
if source_is_sp then
    X[d, 64] = AddPACGA(X[n, 64], SP[]);
else
    X[d, 64] = AddPACGA(X[n, 64], X[m, 64]);
```

C6.2.266 PACIA, PACIA1716, PACIASP, PACIAZ, PACIZA

Pointer Authentication Code for Instruction address, using key A. This instruction computes and inserts a pointer authentication code for an instruction address, using a modifier and key A.

The address is:

- In the general-purpose register that is specified by <Xd> for PACIA and PACIZA.
- In X17, for PACIA1716.
- In X30, for PACIASP and PACIAZ.

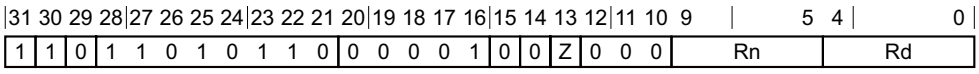
The modifier is:

- In the general-purpose register or stack pointer that is specified by <Xn|SP> for PACIA.
- The value zero, for PACIZA and PACIAZ.
- In X16, for PACIA1716.
- In SP, for PACIASP.

A PACIASP instruction has an implicit BTI instruction. The implicit BTI instruction of a PACIASP instruction is always compatible with PSTATE.BTYPE == 0b01 and PSTATE.BTYPE == 0b10. Controls in SCTLR_ELx configure whether the implicit BTI instruction of a PACIASP instruction is compatible with PSTATE.BTYPE == 0b11. For more information, see *PSTATE.BTYPE*.

Integer

(FEAT_PAuth)



PACIA variant

Applies when Z == 0.

```
PACIA <Xd>, <Xn|SP>
```

PACIZA variant

Applies when Z == 1 && Rn == 11111.

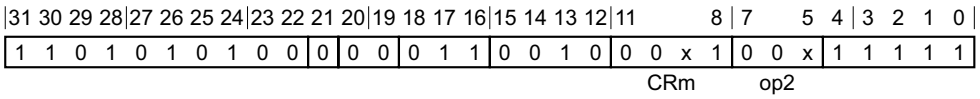
```
PACIZA <Xd>
```

Decode for all variants of this encoding

```
boolean source_is_sp = FALSE;
integer d = UInt(Rd);
integer n = UInt(Rn);

if !IsFeatureImplemented(FEAT_PAuth) then
    UNDEFINED;

if Z == '0' then // PACIA
    if n == 31 then source_is_sp = TRUE;
else // PACIZA
    if n != 31 then UNDEFINED;
```

System

(FEAT_PAuth)



PACIA1716 variant

Applies when `CRm == 0001 && op2 == 000`.

```
PACIA1716
```

PACIASP variant

Applies when `CRm == 0011 && op2 == 001`.

```
PACIASP
```

PACIAZ variant

Applies when `CRm == 0011 && op2 == 000`.

```
PACIAZ
```

Decode for all variants of this encoding

```
integer d;
integer n;
boolean source_is_sp = FALSE;

case CRm:op2 of
    when '0011 000'    // PACIAZ
        d = 30;
        n = 31;
    when '0011 001'    // PACIASP
        d = 30;
        source_is_sp = TRUE;
        if IsFeatureImplemented(FEAT_BTI) then
            // Check for branch target compatibility between PSTATE.BTYPE
            // and implicit branch target of PACIASP instruction.
            SetBTypeCompatible(BTypeCompatible_PACIXSP());
    when '0001 000'    // PACIA1716
        d = 17;
        n = 16;
    when '0001 010' SEE "PACIB";
    when '0001 100' SEE "AUTIA";
    when '0001 110' SEE "AUTIB";
    when '0011 01x' SEE "PACIB";
    when '0011 10x' SEE "AUTIA";
    when '0011 11x' SEE "AUTIB";
    when '0000 111' SEE "XPACLRI";
    otherwise SEE "HINT";
```

Assembler symbols

`<Xd>`	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
`<Xn|SP>`	Is the 64-bit name of the general-purpose source register or stack pointer, encoded in the "Rn" field.

Operation for all encodings

```
if IsFeatureImplemented(FEAT_PAuth) then
    if source_is_sp then
        X[d, 64] = AddPACIA(X[d, 64], SP[]);
    else
        X[d, 64] = AddPACIA(X[d, 64], X[n, 64]);
```

C6.2.267 PACIB, PACIB1716, PACIBSP, PACIBZ, PACIZB

Pointer Authentication Code for Instruction address, using key B. This instruction computes and inserts a pointer authentication code for an instruction address, using a modifier and key B.

The address is:

- In the general-purpose register that is specified by <Xd> for PACIB and PACIZB.
- In X17, for PACIB1716.
- In X30, for PACIBSP and PACIBZ.

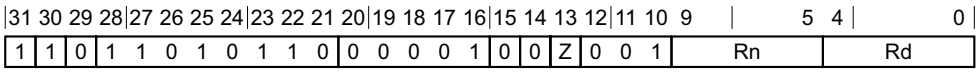
The modifier is:

- In the general-purpose register or stack pointer that is specified by <Xn|SP> for PACIB.
- The value zero, for PACIZB and PACIBZ.
- In X16, for PACIB1716.
- In SP, for PACIBSP.

A PACIBSP instruction has an implicit BTI instruction. The implicit BTI instruction of a PACIBSP instruction is always compatible with PSTATE.BTYPE == 0b01 and PSTATE.BTYPE == 0b10. Controls in SCTLR_ELx configure whether the implicit BTI instruction of a PACIBSP instruction is compatible with PSTATE.BTYPE == 0b11. For more information, see *PSTATE.BTYPE*.

Integer

(FEAT_PAuth)



PACIB variant

Applies when Z == 0.

```
PACIB <Xd>, <Xn|SP>
```

PACIZB variant

Applies when Z == 1 && Rn == 11111.

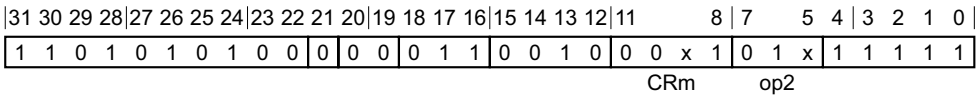
```
PACIZB <Xd>
```

Decode for all variants of this encoding

```
boolean source_is_sp = FALSE;
integer d = UInt(Rd);
integer n = UInt(Rn);

if !IsFeatureImplemented(FEAT_PAuth) then
    UNDEFINED;

if Z == '0' then // PACIB
    if n == 31 then source_is_sp = TRUE;
else // PACIZB
    if n != 31 then UNDEFINED;
```

System

(FEAT_PAuth)



PACIB1716 variant

Applies when `CRm == 0001 && op2 == 010`.

`PACIB1716`

PACIBSP variant

Applies when `CRm == 0011 && op2 == 011`.

`PACIBSP`

PACIBZ variant

Applies when `CRm == 0011 && op2 == 010`.

`PACIBZ`

Decode for all variants of this encoding

```
integer d;
integer n;
boolean source_is_sp = FALSE;

case CRm:op2 of
    when '0011 010'    // PACIBZ
        d = 30;
        n = 31;
    when '0011 011'    // PACIBSP
        d = 30;
        source_is_sp = TRUE;
        if IsFeatureImplemented(FEAT_BTI) then
            // Check for branch target compatibility between PSTATE.BTYPE
            // and implicit branch target of PACIBSP instruction.
            SetBTypeCompatible(BTypeCompatible_PACIXSP());
    when '0001 010'    // PACIB1716
        d = 17;
        n = 16;
    when '0001 000' SEE "PACIA";
    when '0001 100' SEE "AUTIA";
    when '0001 110' SEE "AUTIB";
    when '0011 00x' SEE "PACIA";
    when '0011 10x' SEE "AUTIA";
    when '0011 11x' SEE "AUTIB";
    when '0000 111' SEE "XPACLRI";
    otherwise SEE "HINT";
```

Assembler symbols

`<Xd>`	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
`<Xn|SP>`	Is the 64-bit name of the general-purpose source register or stack pointer, encoded in the "Rn" field.

Operation for all encodings

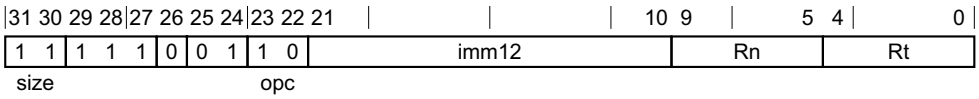
```
if IsFeatureImplemented(FEAT_PAuth) then
    if source_is_sp then
        X[d, 64] = AddPACIB(X[d, 64], SP[]);
    else
        X[d, 64] = AddPACIB(X[d, 64], X[n, 64]);
```

C6.2.268 PRFM (immediate)

Prefetch Memory (immediate) signals the memory system that data memory accesses from a specified address are likely to occur in the near future. The memory system can respond by taking actions that are expected to speed up the memory accesses when they do occur, such as preloading the cache line containing the specified address into one or more caches.

The effect of a PRFM instruction is IMPLEMENTATION DEFINED. For more information, see *Prefetch memory*.

For information about memory accesses, see *Load/store addressing modes*.



Encoding

```
PRFM (<prfop>|#<imm5>), [<Xn|SP>{, #<pimm>}]
```

Decode for this encoding

```
bits(64) offset = LSL(ZeroExtend(imm12, 64), 3);
```

Assembler symbols

<prfop> Is the prefetch operation, defined as <type><target><policy>.

<type> is one of:

PLD Prefetch for load, encoded in the "Rt<4:3>" field as 0b00.

PLI Preload instructions, encoded in the "Rt<4:3>" field as 0b01.

PST Prefetch for store, encoded in the "Rt<4:3>" field as 0b10.

<target> is one of:

L1 Level 1 cache, encoded in the "Rt<2:1>" field as 0b00.

L2 Level 2 cache, encoded in the "Rt<2:1>" field as 0b01.

L3 Level 3 cache, encoded in the "Rt<2:1>" field as 0b10.

SLC When FEAT_PRFMSLC is implemented, system level cache, encoded in the "Rt<2:1>" field as 0b11.

<policy> is one of:

KEEP Retained or temporal prefetch, allocated in the cache normally. Encoded in the "Rt<0>" field as 0.

STRM Streaming or non-temporal prefetch, for data that is used only once. Encoded in the "Rt<0>" field as 1.

For more information on these prefetch operations, see *Prefetch memory*.

For other encodings of the "Rt" field, use <imm5>.

<imm5> Is the prefetch operation encoding as an immediate, in the range 0 to 31, encoded in the "Rt" field.

This syntax is only for encodings that are not accessible using <prfop>.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<pimm> Is the optional positive immediate byte offset, a multiple of 8 in the range 0 to 32760, defaulting to 0 and encoded in the "imm12" field as <pimm>/8.

Shared decode for all encodings

```
integer n = UInt(Rn);
integer t = UInt(Rt);
```

Operation

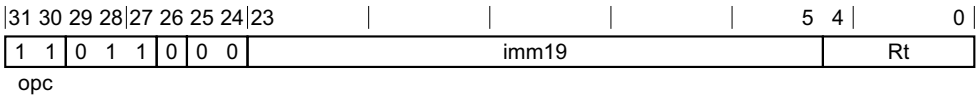
```
bits(64) address;

boolean privileged = PSTATE.EL != EL0;
AccessDescriptor accdesc = CreateAccDescGPR(MemOp_PREFETCH, FALSE, privileged, FALSE);

if n == 31 then
    address = SP[];
else
    address = X[n, 64];

address = GenerateAddress(address, offset, accdesc);

Prefetch(address, t<4:0>);
```

C6.2.269 PRFM (literal)

Prefetch Memory (literal) signals the memory system that data memory accesses from a specified address are likely to occur in the near future. The memory system can respond by taking actions that are expected to speed up the memory accesses when they do occur, such as preloading the cache line containing the specified address into one or more caches.

The effect of a PRFM instruction is IMPLEMENTATION DEFINED. For more information, see *Prefetch memory*.

For information about memory accesses, see *Load/store addressing modes*.



Encoding

```
PRFM (<prfop>|#<imm5>), <label>
```

Decode for this encoding

```
integer t = UInt(Rt);

bits(64) offset = SignExtend(imm19:'00', 64);
```

Assembler symbols

<prfop> Is the prefetch operation, defined as <type><target><policy>.

<type> is one of:

PLD Prefetch for load, encoded in the "Rt<4:3>" field as 0b00.

PLI Preload instructions, encoded in the "Rt<4:3>" field as 0b01.

PST Prefetch for store, encoded in the "Rt<4:3>" field as 0b10.

<target> is one of:

L1 Level 1 cache, encoded in the "Rt<2:1>" field as 0b00.

L2 Level 2 cache, encoded in the "Rt<2:1>" field as 0b01.

L3 Level 3 cache, encoded in the "Rt<2:1>" field as 0b10.

SLC When FEAT_PRFMSLC is implemented, system level cache, encoded in the "Rt<2:1>" field as 0b11.

<policy> is one of:

KEEP Retained or temporal prefetch, allocated in the cache normally. Encoded in the "Rt<0>" field as 0.

STRM Streaming or non-temporal prefetch, for data that is used only once. Encoded in the "Rt<0>" field as 1.

For more information on these prefetch operations, see *Prefetch memory*.

For other encodings of the "Rt" field, use <imm5>.

<imm5> Is the prefetch operation encoding as an immediate, in the range 0 to 31, encoded in the "Rt" field.

This syntax is only for encodings that are not accessible using <prfop>.

<label> Is the program label from which the data is to be loaded. Its offset from the address of this instruction, in the range +/-1MB, is encoded as "imm19" times 4.

Operation

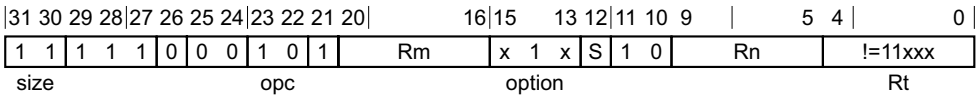
```
bits(64) address = PC64 + offset;

Prefetch(address, t<4:0>);
```

C6.2.270 PRFM (register)

Prefetch Memory (register) signals the memory system that data memory accesses from a specified address are likely to occur in the near future. The memory system can respond by taking actions that are expected to speed up the memory accesses when they do occur, such as preloading the cache line containing the specified address into one or more caches.

The effect of a PRFM instruction is IMPLEMENTATION DEFINED. For more information, see *Prefetch memory*.

For information about memory accesses, see *Load/store addressing modes*.



Encoding

```
PRFM (<prfop>|#<imm5>), [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

Decode for this encoding

```
if option<1> == '0' then UNDEFINED;    // sub-word index
ExtendType extend_type = DecodeRegExtend(option);
integer shift = if S == '1' then 3 else 0;
```

Assembler symbols

<prfop> Is the prefetch operation, defined as <type><target><policy>.

<type> is one of:

PLD Prefetch for load, encoded in the "Rt<4:3>" field as 0b00.

PLI Preload instructions, encoded in the "Rt<4:3>" field as 0b01.

PST Prefetch for store, encoded in the "Rt<4:3>" field as 0b10.

<target> is one of:

L1 Level 1 cache, encoded in the "Rt<2:1>" field as 0b00.

L2 Level 2 cache, encoded in the "Rt<2:1>" field as 0b01.

L3 Level 3 cache, encoded in the "Rt<2:1>" field as 0b10.

SLC When FEAT_PRFMSLC is implemented, system level cache, encoded in the "Rt<2:1>" field as 0b11.

<policy> is one of:

KEEP Retained or temporal prefetch, allocated in the cache normally. Encoded in the "Rt<0>" field as 0.

STRM Streaming or non-temporal prefetch, for data that is used only once. Encoded in the "Rt<0>" field as 1.

For more information on these prefetch operations, see *Prefetch memory*.

For other encodings of the "Rt" field, use <imm5>.

<imm5> Is the prefetch operation encoding as an immediate, in the range 0 to 31, encoded in the "Rt" field.

This syntax is only for encodings that are not accessible using <prfop>.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Wm> When option<0> is set to 0, is the 32-bit name of the general-purpose index register, encoded in the "Rm" field.

<Xm> When option<0> is set to 1, is the 64-bit name of the general-purpose index register, encoded in the "Rm" field.

<extend> Is the index extend/shift specifier, defaulting to LSL, and which must be omitted for the LSL option when <amount> is omitted. encoded in the "option" field. It can have the following values:

- UXTW when option = 010
- LSL when option = 011
- SXTW when option = 110
- SXTX when option = 111

<amount> Is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in the "S" field. It can have the following values:

- #0 when S = 0
- #3 when S = 1

Shared decode for all encodings

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer m = UInt(Rm);
```

Operation

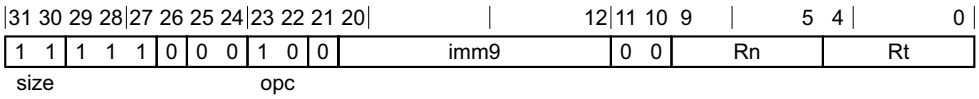
```
bits(64) offset = ExtendReg(m, extend_type, shift, 64);
bits(64) address;

boolean privileged = PSTATE.EL != EL0;
AccessDescriptor accdesc = CreateAccDescGPR(MemOp_PREFETCH, FALSE, privileged, FALSE);

if n == 31 then
    address = SP[];
else
    address = X[n, 64];

address = GenerateAddress(address, offset, accdesc);

Prefetch(address, t<4:0>);
```

C6.2.271 PRFUM

Prefetch Memory (unscaled offset) signals the memory system that data memory accesses from a specified address are likely to occur in the near future. The memory system can respond by taking actions that are expected to speed up the memory accesses when they do occur, such as preloading the cache line containing the specified address into one or more caches.

The effect of a PRFUM instruction is IMPLEMENTATION DEFINED. For more information, see *Prefetch memory*.

For information about memory accesses, see *Load/store addressing modes*.



Encoding

```
PRFUM (<prfop>|#<imm5>), [<Xn|SP>{, #<simm>}]
```

Decode for this encoding

```
bits(64) offset = SignExtend(imm9, 64);
```

Assembler symbols

<prfop> Is the prefetch operation, defined as <type><target><policy>.

<type> is one of:

PLD Prefetch for load, encoded in the "Rt<4:3>" field as 0b00.

PLI Preload instructions, encoded in the "Rt<4:3>" field as 0b01.

PST Prefetch for store, encoded in the "Rt<4:3>" field as 0b10.

<target> is one of:

L1 Level 1 cache, encoded in the "Rt<2:1>" field as 0b00.

L2 Level 2 cache, encoded in the "Rt<2:1>" field as 0b01.

L3 Level 3 cache, encoded in the "Rt<2:1>" field as 0b10.

<policy> is one of:

KEEP Retained or temporal prefetch, allocated in the cache normally. Encoded in the "Rt<0>" field as 0.

STRM Streaming or non-temporal prefetch, for data that is used only once. Encoded in the "Rt<0>" field as 1.

For more information on these prefetch operations, see *Prefetch memory*.

For other encodings of the "Rt" field, use <imm5>.

<imm5> Is the prefetch operation encoding as an immediate, in the range 0 to 31, encoded in the "Rt" field.

This syntax is only for encodings that are not accessible using <prfop>.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<simm> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared decode for all encodings

```
integer n = UInt(Rn);
integer t = UInt(Rt);
```

Operation

```
bits(64) address;

boolean privileged = PSTATE.EL != EL0;
AccessDescriptor accdesc = CreateAccDescGPR(MemOp_PREFETCH, FALSE, privileged, FALSE);

if n == 31 then
    address = SP[];
else
    address = X[n, 64];

address = GenerateAddress(address, offset, accdesc);

Prefetch(address, t<4:0>);
```

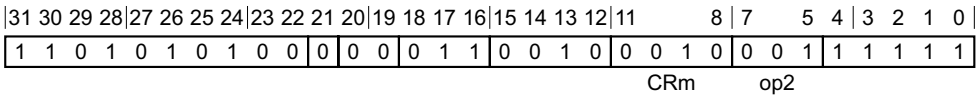
C6.2.272 PSB

Profiling Synchronization Barrier. This instruction is a barrier that ensures that all existing profiling data for the current PE has been formatted, and profiling buffer addresses have been translated such that all writes to the profiling buffer have been initiated. A following `DSB` instruction completes when the writes to the profiling buffer have completed.

If FEAT_SPE is not implemented, this instruction executes as a `NOP`.

System

(FEAT_SPE)



Encoding

```
PSB CSYNC
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SPE) then EndOfInstruction();
```

Operation

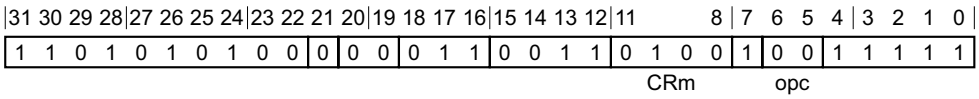
```
ProfilingSynchronizationBarrier();
```

C6.2.273 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)*.

This instruction is an alias of the DSB instruction. This means that:

- The encodings in this description are named to match the encodings of DSB.
- The description of DSB gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.



Encoding

PSSBB

is equivalent to

DSB #4

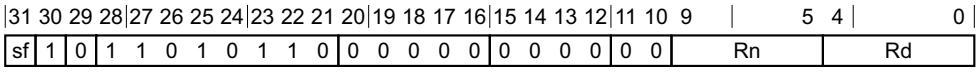
and is always the preferred disassembly.

Operation

The description of DSB gives the operational pseudocode for this instruction.

C6.2.274 RBIT

Reverse Bits reverses the bit order in a register.



32-bit variant

Applies when `sf == 0`.

```
RBIT <Wd>, <Wn>
```

64-bit variant

Applies when `sf == 1`.

```
RBIT <Xd>, <Xn>
```

Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

constant integer datasize = 32 << UInt(sf);
```

Assembler symbols

`<Wd>`	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
`<Wn>`	Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
`<Xd>`	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
`<Xn>`	Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.

Operation

```
bits(datasize) operand = X[n, datasize];
bits(datasize) result;

for i = 0 to datasize-1
    result<(datasize-1)-i> = operand<i>;

X[d, datasize] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.

— The values of the NZCV flags.

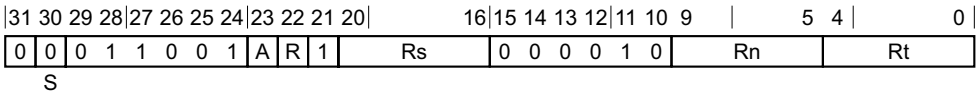
C6.2.275 RCWCAS, RCWCASA, RCWCASL, RCWCASAL

Read Check Write Compare and Swap doubleword in memory reads a 64-bit doubleword from memory, and compares it against the value held in a register. If the comparison is equal, the value in a second register is conditionally written to memory. Storing back to memory is conditional on RCW Checks. If the write is performed, the read and the write occur atomically such that no other modification of the memory location can take place between the read and the write. This instruction updates the condition flags based on the result of the update of memory.

- RCWCASA and RCWCASAL load from memory with acquire semantics.
- RCWCASL and RCWCASAL store to memory with release semantics.
- RCWCAS has neither acquire nor release semantics.

Integer

(FEAT_THE)



RCWCAS variant

Applies when A == 0 && R == 0.

```
RCWCAS <Xs>, <Xt>, [<Xn|SP>]
```

RCWCASA variant

Applies when A == 1 && R == 0.

```
RCWCASA <Xs>, <Xt>, [<Xn|SP>]
```

RCWCASAL variant

Applies when A == 1 && R == 1.

```
RCWCASAL <Xs>, <Xt>, [<Xn|SP>]
```

RCWCASL variant

Applies when A == 0 && R == 1.

```
RCWCASL <Xs>, <Xt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_THE) then UNDEFINED;
integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

boolean acquire = A == '1';
boolean release = R == '1';
boolean tagchecked = n != 31;
```

Assembler symbols

<Xs> Is the 64-bit name of the general-purpose register to be compared and loaded, encoded in the "Rs" field.

<Xt> Is the 64-bit name of the general-purpose register to be conditionally stored, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
if IsD128Enabled(PSTATE.EL) then UNDEFINED;
bits(64) address;
bits(64) newdata = X[t, 64];
bits(64) compdata = X[s, 64];
bits(64) readdata;
bits(4) nzcv;

AccessDescriptor accdesc = CreateAccDescRCW(MemAtomicOp_CAS, FALSE, acquire, release, tagchecked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

(nzcv, readdata) = MemAtomicRCW(address, compdata, newdata, accdesc);

PSTATE.<N,Z,C,V> = nzcv;
X[s, 64] = readdata;    // Return the old value when s!=31
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

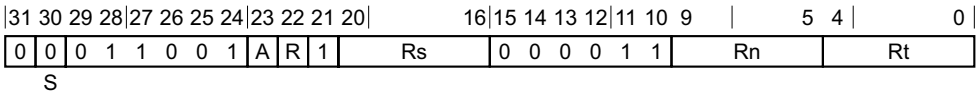
C6.2.276 RCWCASP, RCWCASPA, RCWCASPL, RCWCASPAL

Read Check Write Compare and Swap quadword in memory reads a 128-bit quadword from memory, and compares it against the value held in a pair of registers. If the comparison is equal, the value in a second pair of registers is conditionally written to memory. Storing back to memory is conditional on RCW Checks. If the write is performed, the read and the write occur atomically such that no other modification of the memory location can take place between the read and the write. This instruction updates the condition flags based on the result of the update of memory.

- RCWCASPA and RCWCASPAL load from memory with acquire semantics.
- RCWCASPL and RCWCASPAL store to memory with release semantics.
- RCWCASP has neither acquire nor release semantics.

Integer

(FEAT_D128 && FEAT_THE)



RCWCASP variant

Applies when A == 0 && R == 0.

```
RCWCASP <Xs>, <X(s+1)>, <Xt>, <X(t+1)>, [<Xn|SP>]
```

RCWCASPA variant

Applies when A == 1 && R == 0.

```
RCWCASPA <Xs>, <X(s+1)>, <Xt>, <X(t+1)>, [<Xn|SP>]
```

RCWCASPAL variant

Applies when A == 1 && R == 1.

```
RCWCASPAL <Xs>, <X(s+1)>, <Xt>, <X(t+1)>, [<Xn|SP>]
```

RCWCASPL variant

Applies when A == 0 && R == 1.

```
RCWCASPL <Xs>, <X(s+1)>, <Xt>, <X(t+1)>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_D128) || !IsFeatureImplemented(FEAT_THE) then UNDEFINED;
if Rs<0> == '1' then UNDEFINED;
if Rt<0> == '1' then UNDEFINED;
integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

boolean acquire = A == '1';
boolean release = R == '1';
boolean tagchecked = n != 31;
```

Assembler symbols

<Xs>	Is the 64-bit name of the first general-purpose register to be compared and loaded, encoded in the "Rs" field. <Xs> must be an even-numbered register.
<X(s+1)>	Is the 64-bit name of the second general-purpose register to be compared and loaded.
<Xt>	Is the 64-bit name of the first general-purpose register to be conditionally stored, encoded in the "Rt" field. <Xt> must be an even-numbered register.
<X(t+1)>	Is the 64-bit name of the second general-purpose register to be conditionally stored.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
if !IsD128Enabled(PSTATE.EL) then UNDEFINED;
bits(64) address;
bits(128) newdata;
bits(128) compdata;
bits(128) readdata;
bits(4) nzcv;

bits(64) s1 = X[s, 64];
bits(64) s2 = X[s+1, 64];
bits(64) t1 = X[t, 64];
bits(64) t2 = X[t+1, 64];

AccessDescriptor accdesc = CreateAccDescRCW(MemAtomicOp_CAS, FALSE, acquire, release, tagchecked);

compdata = if BigEndian(accdesc.acctype) then s1:s2 else s2:s1;
newdata = if BigEndian(accdesc.acctype) then t1:t2 else t2:t1;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

(nzcv, readdata) = MemAtomicRCW(address, compdata, newdata, accdesc);

PSTATE.<N,Z,C,V> = nzcv;
if BigEndian(accdesc.acctype) then
    X[s, 64] = readdata<127:64>;
    X[s+1, 64] = readdata<63:0>;
else
    X[s, 64] = readdata<63:0>;
    X[s+1, 64] = readdata<127:64>;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

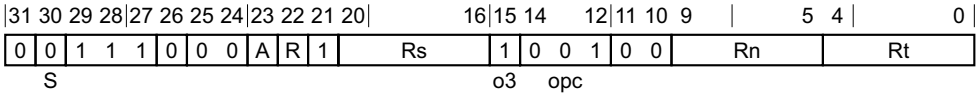
C6.2.277 RCWCLR, RCWCLRA, RCWCLRL, RCWCLRAL

Read Check Write atomic bit Clear on doubleword in memory atomically loads a 64-bit doubleword from memory, performs a bitwise AND with the complement of the value held in a register on it, and conditionally stores the result back to memory. Storing of the result back to memory is conditional on RCW Checks. The value initially loaded from memory is returned in the destination register. This instruction updates the condition flags based on the result of the update of memory.

- `RCWCLRA` and `RCWCLRAL` load from memory with acquire semantics.
- `RCWCLRL` and `RCWCLRAL` store to memory with release semantics.
- `RCWCLR` has neither acquire nor release semantics.

Integer

(FEAT_THE)



RCWCLR variant

Applies when `A == 0 && R == 0`.

```
RCWCLR <Xs>, <Xt>, [<Xn|SP>]
```

RCWCLRA variant

Applies when `A == 1 && R == 0`.

```
RCWCLRA <Xs>, <Xt>, [<Xn|SP>]
```

RCWCLRAL variant

Applies when `A == 1 && R == 1`.

```
RCWCLRAL <Xs>, <Xt>, [<Xn|SP>]
```

RCWCLRL variant

Applies when `A == 0 && R == 1`.

```
RCWCLRL <Xs>, <Xt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_THE) then UNDEFINED;
integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

boolean acquire = A == '1' && Rt != '11111';
boolean release = R == '1';
boolean tagchecked = n != 31;
```

Assembler symbols

<Xs> Is the 64-bit name of the general-purpose register to be stored, encoded in the "Rs" field.

<Xt> Is the 64-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
if IsD128Enabled(PSTATE.EL) then UNDEFINED;
bits(64) address;
bits(64) newdata = X[s, 64];
bits(64) readdata;
bits(4) nzcv;

AccessDescriptor accdesc = CreateAccDescRCW(MemAtomicOp_BIC, FALSE, acquire, release, tagchecked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

bits(64) compdata = bits(64) UNKNOWN;    // Irrelevant when not executing CAS
(nzcv, readdata) = MemAtomicRCW(address, compdata, newdata, accdesc);

PSTATE.<N,Z,C,V> = nzcv;
X[t, 64] = readdata;    // Return the old value when t!=31
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

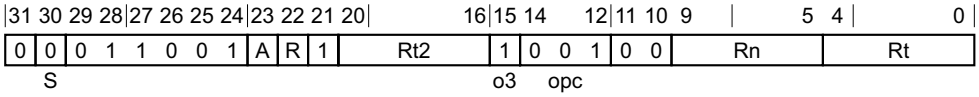
C6.2.278 RCWCLRP, RCWCLRPA, RCWCLRPL, RCWCLRPAL

Read Check Write atomic bit Clear on quadword in memory atomically loads a 128-bit quadword from memory, performs a bitwise AND with the complement of the value held in a pair of registers on it, and conditionally stores the result back to memory. Storing of the result back to memory is conditional on RCW Checks. The value initially loaded from memory is returned in the same pair of registers. This instruction updates the condition flags based on the result of the update of memory.

- RCWCLRPA and RCWCLRPAL load from memory with acquire semantics.
- RCWCLRPL and RCWCLRPAL store to memory with release semantics.
- RCWCLRP has neither acquire nor release semantics.

Integer

(FEAT_D128 && FEAT_THE)



RCWCLRP variant

Applies when A == 0 && R == 0.

```
RCWCLRP <Xt1>, <Xt2>, [<Xn|SP>]
```

RCWCLRPA variant

Applies when A == 1 && R == 0.

```
RCWCLRPA <Xt1>, <Xt2>, [<Xn|SP>]
```

RCWCLRPAL variant

Applies when A == 1 && R == 1.

```
RCWCLRPAL <Xt1>, <Xt2>, [<Xn|SP>]
```

RCWCLRPL variant

Applies when A == 0 && R == 1.

```
RCWCLRPL <Xt1>, <Xt2>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_D128) || !IsFeatureImplemented(FEAT_THE) then UNDEFINED;
if Rt == '11111' then UNDEFINED;
if Rt2 == '11111' then UNDEFINED;
integer t = UInt(Rt);
integer t2 = UInt(Rt2);
integer n = UInt(Rn);

boolean acquire = A == '1';
boolean release = R == '1';
boolean tagchecked = n != 31;

boolean rt_unknown = FALSE;

if t == t2 then
    Constraint c = ConstrainUnpredictable(Unpredictable_LSE128OVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
```

```
        when Constraint_UNKNOWN rt_unknown = TRUE;    // result is UNKNOWN
        when Constraint_UNDEF   UNDEFINED;
        when Constraint_NOP     EndOfInstruction();
```

Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*, and particularly *CONSTRAINED UNPREDICTABLE behavior for A64 instructions*.

Assembler symbols

<Xt1>	Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.
<Xt2>	Is the 64-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
if !IsD128Enabled(PSTATE.EL) then UNDEFINED;
bits(64) address;
bits(64) value1;
bits(64) value2;
bits(128) newdata;
bits(128) readdata;
bits(4) nzcv;

AccessDescriptor accdesc = CreateAccDescRCW(MemAtomicOp_BIC, FALSE, acquire, release, tagchecked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

value1 = X[t, 64];
value2 = X[t2, 64];

newdata = if BigEndian(accdesc.acctype) then value1:value2 else value2:value1;

bits(128) compdata = bits(128) UNKNOWN;    // Irrelevant when not executing CAS
(nzcv, readdata) = MemAtomicRCW(address, compdata, newdata, accdesc);

PSTATE.<N,Z,C,V> = nzcv;
if rt_unknown then
    readdata = bits(128) UNKNOWN;

if BigEndian(accdesc.acctype) then
    X[t, 64] = readdata<127:64>;
    X[t2, 64] = readdata<63:0>;
else
    X[t, 64] = readdata<63:0>;
    X[t2, 64] = readdata<127:64>;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

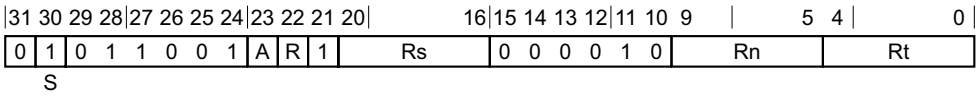
C6.2.279 RCWSCAS, RCWSCASA, RCWSCASL, RCWSCASAL

Read Check Write Software Compare and Swap doubleword in memory reads a 64-bit doubleword from memory, and compares it against the value held in a register. If the comparison is equal, the value in a second register is conditionally written to memory. Storing back to memory is conditional on RCW Checks and RCWS Checks. If the write is performed, the read and the write occur atomically such that no other modification of the memory location can take place between the read and the write. This instruction updates the condition flags based on the result of the update of memory.

- RCWSCASA and RCWSCASAL load from memory with acquire semantics.
- RCWSCASL and RCWSCASAL store to memory with release semantics.
- RCWSCAS has neither acquire nor release semantics.

Integer

(FEAT_THE)



RCWSCAS variant

Applies when A == 0 && R == 0.

```
RCWSCAS <Xs>, <Xt>, [<Xn|SP>]
```

RCWSCASA variant

Applies when A == 1 && R == 0.

```
RCWSCASA <Xs>, <Xt>, [<Xn|SP>]
```

RCWSCASAL variant

Applies when A == 1 && R == 1.

```
RCWSCASAL <Xs>, <Xt>, [<Xn|SP>]
```

RCWSCASL variant

Applies when A == 0 && R == 1.

```
RCWSCASL <Xs>, <Xt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_THE) then UNDEFINED;
integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

boolean acquire = A == '1';
boolean release = R == '1';
boolean tagchecked = n != 31;
```

Assembler symbols

<Xs> Is the 64-bit name of the general-purpose register to be compared and loaded, encoded in the "Rs" field.

<Xt> Is the 64-bit name of the general-purpose register to be conditionally stored, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
if IsD128Enabled(PSTATE.EL) then UNDEFINED;
bits(64) address;
bits(64) newdata = X[t, 64];
bits(64) compdata = X[s, 64];
bits(64) readdata;
bits(4) nzcv;

AccessDescriptor accdesc = CreateAccDescRCW(MemAtomicOp_CAS, TRUE, acquire, release, tagchecked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

(nzcv, readdata) = MemAtomicRCW(address, compdata, newdata, accdesc);

PSTATE.<N,Z,C,V> = nzcv;
X[s, 64] = readdata;    // Return the old value when s!=31
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

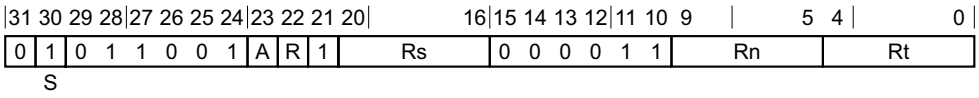
C6.2.280 RCWSCASP, RCWSCASPA, RCWSCASPL, RCWSCASPAL

Read Check Write Software Compare and Swap quadword in memory reads a 128-bit quadword from memory, and compares it against the value held in a pair of registers. If the comparison is equal, the value in a second pair of registers is conditionally written to memory. Storing back to memory is conditional on RCW Checks and RCWS Checks. If the write is performed, the read and the write occur atomically such that no other modification of the memory location can take place between the read and the write. This instruction updates the condition flags based on the result of the update of memory.

- RCWSCASPA and RCWSCASPAL load from memory with acquire semantics.
- RCWSCASPL and RCWSCASPAL store to memory with release semantics.
- RCWSCASP has neither acquire nor release semantics.

Integer

(FEAT_D128 && FEAT_THE)



RCWSCASP variant

Applies when A == 0 && R == 0.

```
RCWSCASP <Xs>, <X(s+1)>, <Xt>, <X(t+1)>, [<Xn|SP>]
```

RCWSCASPA variant

Applies when A == 1 && R == 0.

```
RCWSCASPA <Xs>, <X(s+1)>, <Xt>, <X(t+1)>, [<Xn|SP>]
```

RCWSCASPAL variant

Applies when A == 1 && R == 1.

```
RCWSCASPAL <Xs>, <X(s+1)>, <Xt>, <X(t+1)>, [<Xn|SP>]
```

RCWSCASPL variant

Applies when A == 0 && R == 1.

```
RCWSCASPL <Xs>, <X(s+1)>, <Xt>, <X(t+1)>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_D128) || !IsFeatureImplemented(FEAT_THE) then UNDEFINED;
if Rs<0> == '1' then UNDEFINED;
if Rt<0> == '1' then UNDEFINED;
integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

boolean acquire = A == '1';
boolean release = R == '1';
boolean tagchecked = n != 31;
```

Assembler symbols

<Xs>	Is the 64-bit name of the first general-purpose register to be compared and loaded, encoded in the "Rs" field. <Xs> must be an even-numbered register.
<X(s+1)>	Is the 64-bit name of the second general-purpose register to be compared and loaded.
<Xt>	Is the 64-bit name of the first general-purpose register to be conditionally stored, encoded in the "Rt" field. <Xt> must be an even-numbered register.
<X(t+1)>	Is the 64-bit name of the second general-purpose register to be conditionally stored.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
if !IsD128Enabled(PSTATE.EL) then UNDEFINED;
bits(64) address;
bits(128) newdata;
bits(128) compdata;
bits(128) readdata;
bits(4) nzcv;

bits(64) s1 = X[s, 64];
bits(64) s2 = X[s+1, 64];
bits(64) t1 = X[t, 64];
bits(64) t2 = X[t+1, 64];

AccessDescriptor accdesc = CreateAccDescRCW(MemAtomicOp_CAS, TRUE, acquire, release, tagchecked);

compdata = if BigEndian(accdesc.acctype) then s1:s2 else s2:s1;
newdata = if BigEndian(accdesc.acctype) then t1:t2 else t2:t1;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

(nzcv, readdata) = MemAtomicRCW(address, compdata, newdata, accdesc);

PSTATE.<N,Z,C,V> = nzcv;
if BigEndian(accdesc.acctype) then
    X[s, 64] = readdata<127:64>;
    X[s+1, 64] = readdata<63:0>;
else
    X[s, 64] = readdata<63:0>;
    X[s+1, 64] = readdata<127:64>;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

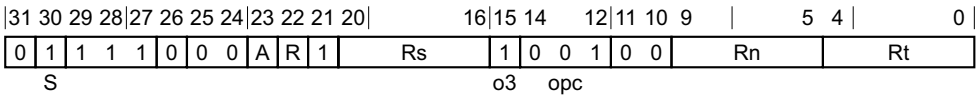
C6.2.281 RCWSCLR, RCWSCLRA, RCWSCLRL, RCWSCLRAL

Read Check Write Software atomic bit Clear on doubleword in memory atomically loads a 64-bit doubleword from memory, performs a bitwise AND with the complement of the value held in a register on it, and conditionally stores the result back to memory. Storing of the result back to memory is conditional on RCW Checks and RCWS Checks. The value initially loaded from memory is returned in the destination register. This instruction updates the condition flags based on the result of the update of memory.

- RCWSCLRA and RCWSCLRAL load from memory with acquire semantics.
- RCWSCLRL and RCWSCLRAL store to memory with release semantics.
- RCWSCLR has neither acquire nor release semantics.

Integer

(FEAT_THE)



RCWSCLR variant

Applies when A == 0 && R == 0.

```
RCWSCLR <Xs>, <Xt>, [<Xn|SP>]
```

RCWSCLRA variant

Applies when A == 1 && R == 0.

```
RCWSCLRA <Xs>, <Xt>, [<Xn|SP>]
```

RCWSCLRAL variant

Applies when A == 1 && R == 1.

```
RCWSCLRAL <Xs>, <Xt>, [<Xn|SP>]
```

RCWSCLRL variant

Applies when A == 0 && R == 1.

```
RCWSCLRL <Xs>, <Xt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_THE) then UNDEFINED;
integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

boolean acquire = A == '1' && Rt != '11111';
boolean release = R == '1';
boolean tagchecked = n != 31;
```

Assembler symbols

<Xs> Is the 64-bit name of the general-purpose register to be stored, encoded in the "Rs" field.

<Xt> Is the 64-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
if IsD128Enabled(PSTATE.EL) then UNDEFINED;
bits(64) address;
bits(64) newdata = X[s, 64];
bits(64) readdata;
bits(4) nzcv;

AccessDescriptor accdesc = CreateAccDescRCW(MemAtomicOp_BIC, TRUE, acquire, release, tagchecked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

bits(64) compdata = bits(64) UNKNOWN;    // Irrelevant when not executing CAS
(nzcv, readdata) = MemAtomicRCW(address, compdata, newdata, accdesc);

PSTATE.<N,Z,C,V> = nzcv;
X[t, 64] = readdata;    // Return the old value when t!=31
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

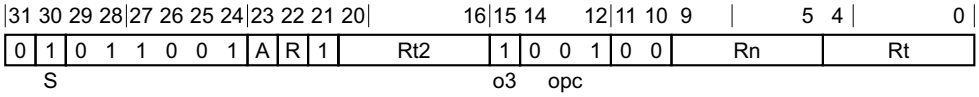
C6.2.282 RCWSCLRP, RCWSCLRPA, RCWSCLRPL, RCWSCLRPAL

Read Check Write Software atomic bit Clear on quadword in memory atomically loads a 128-bit quadword from memory, performs a bitwise AND with the complement of the value held in a pair of registers on it, and conditionally stores the result back to memory. Storing of the result back to memory is conditional on RCW Checks and RCWS Checks. The value initially loaded from memory is returned in the same pair of registers. This instruction updates the condition flags based on the result of the update of memory.

- RCWSCLRPA and RCWSCLRPAL load from memory with acquire semantics.
- RCWSCLRPL and RCWSCLRPAL store to memory with release semantics.
- RCWSCLRP has neither acquire nor release semantics.

Integer

(FEAT_D128 && FEAT_THE)



RCWSCLRP variant

Applies when A == 0 && R == 0.

```
RCWSCLRP <Xt1>, <Xt2>, [<Xn|SP>]
```

RCWSCLRPA variant

Applies when A == 1 && R == 0.

```
RCWSCLRPA <Xt1>, <Xt2>, [<Xn|SP>]
```

RCWSCLRPAL variant

Applies when A == 1 && R == 1.

```
RCWSCLRPAL <Xt1>, <Xt2>, [<Xn|SP>]
```

RCWSCLRPL variant

Applies when A == 0 && R == 1.

```
RCWSCLRPL <Xt1>, <Xt2>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_D128) || !IsFeatureImplemented(FEAT_THE) then UNDEFINED;
if Rt == '11111' then UNDEFINED;
if Rt2 == '11111' then UNDEFINED;
integer t = UInt(Rt);
integer t2 = UInt(Rt2);
integer n = UInt(Rn);

boolean acquire = A == '1';
boolean release = R == '1';
boolean tagchecked = n != 31;

boolean rt_unknown = FALSE;

if t == t2 then
    Constraint c = ConstrainUnpredictable(Unpredictable_LSE128OVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
```

```
when Constraint_UNKNOWN rt_unknown = TRUE;    // result is UNKNOWN
when Constraint_UNDEF   UNDEFINED;
when Constraint_NOP     EndOfInstruction();
```

Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*, and particularly *CONSTRAINED UNPREDICTABLE behavior for A64 instructions*.

Assembler symbols

<Xt1>	Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.
<Xt2>	Is the 64-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
if !IsD128Enabled(PSTATE.EL) then UNDEFINED;
bits(64) address;
bits(64) value1;
bits(64) value2;
bits(128) newdata;
bits(128) readdata;
bits(4) nzcv;

AccessDescriptor accdesc = CreateAccDescRCW(MemAtomicOp_BIC, TRUE, acquire, release, tagchecked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

value1 = X[t, 64];
value2 = X[t2, 64];

newdata = if BigEndian(accdesc.acctype) then value1:value2 else value2:value1;

bits(128) compdata = bits(128) UNKNOWN;    // Irrelevant when not executing CAS
(nzcv, readdata) = MemAtomicRCW(address, compdata, newdata, accdesc);

PSTATE.<N,Z,C,V> = nzcv;
if rt_unknown then
    readdata = bits(128) UNKNOWN;

if BigEndian(accdesc.acctype) then
    X[t, 64] = readdata<127:64>;
    X[t2, 64] = readdata<63:0>;
else
    X[t, 64] = readdata<63:0>;
    X[t2, 64] = readdata<127:64>;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

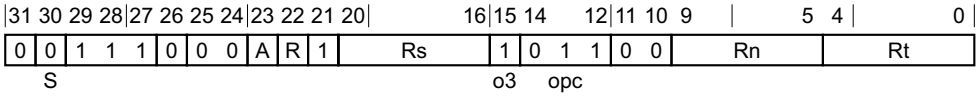
C6.2.283 RCWSET, RCWSETA, RCWSETL, RCWSETAL

Read Check Write atomic bit Set on doubleword in memory atomically loads a 64-bit doubleword from memory, performs a bitwise OR with the complement of the value held in a register on it, and conditionally stores the result back to memory. Storing of the result back to memory is conditional on RCW Checks. The value initially loaded from memory is returned in the destination register. This instruction updates the condition flags based on the result of the update of memory.

- RCWSETA and RCWSETAL load from memory with acquire semantics.
- RCWSETL and RCWSETAL store to memory with release semantics.
- RCWSET has neither acquire nor release semantics.

Integer

(FEAT_THE)



RCWSET variant

Applies when A == 0 && R == 0.

```
RCWSET <Xs>, <Xt>, [<Xn|SP>]
```

RCWSETA variant

Applies when A == 1 && R == 0.

```
RCWSETA <Xs>, <Xt>, [<Xn|SP>]
```

RCWSETAL variant

Applies when A == 1 && R == 1.

```
RCWSETAL <Xs>, <Xt>, [<Xn|SP>]
```

RCWSETL variant

Applies when A == 0 && R == 1.

```
RCWSETL <Xs>, <Xt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_THE) then UNDEFINED;
integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

boolean acquire = A == '1' && Rt != '11111';
boolean release = R == '1';
boolean tagchecked = n != 31;
```

Assembler symbols

<Xs> Is the 64-bit name of the general-purpose register to be stored, encoded in the "Rs" field.

<Xt> Is the 64-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
if IsD128Enabled(PSTATE.EL) then UNDEFINED;
bits(64) address;
bits(64) newdata = X[s, 64];
bits(64) readdata;
bits(4) nzcv;

AccessDescriptor accdesc = CreateAccDescRCW(MemAtomicOp_ORR, FALSE, acquire, release, tagchecked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

bits(64) compdata = bits(64) UNKNOWN;    // Irrelevant when not executing CAS
(nzcv, readdata) = MemAtomicRCW(address, compdata, newdata, accdesc);

PSTATE.<N,Z,C,V> = nzcv;
X[t, 64] = readdata;    // Return the old value when t!=31
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

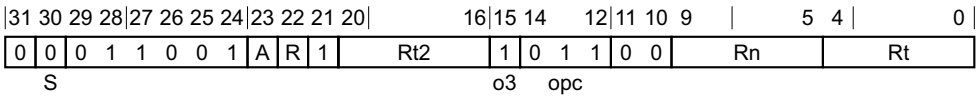
C6.2.284 RCWSETP, RCWSETPA, RCWSETPL, RCWSETPAL

Read Check Write atomic bit Set on quadword in memory atomically loads a 128-bit quadword from memory, performs a bitwise OR with the value held in a pair of registers on it, and conditionally stores the result back to memory. Storing of the result back to memory is conditional on RCW Checks. The value initially loaded from memory is returned in the same pair of registers. This instruction updates the condition flags based on the result of the update of memory.

- RCWSETPA and RCWSETPAL load from memory with acquire semantics.
- RCWSETPL and RCWSETPAL store to memory with release semantics.
- RCWSETP has neither acquire nor release semantics.

Integer

(FEAT_D128 && FEAT_THE)



RCWSETP variant

Applies when A == 0 && R == 0.

```
RCWSETP <Xt1>, <Xt2>, [<Xn|SP>]
```

RCWSETPA variant

Applies when A == 1 && R == 0.

```
RCWSETPA <Xt1>, <Xt2>, [<Xn|SP>]
```

RCWSETPAL variant

Applies when A == 1 && R == 1.

```
RCWSETPAL <Xt1>, <Xt2>, [<Xn|SP>]
```

RCWSETPL variant

Applies when A == 0 && R == 1.

```
RCWSETPL <Xt1>, <Xt2>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_D128) || !IsFeatureImplemented(FEAT_THE) then UNDEFINED;
if Rt == '11111' then UNDEFINED;
if Rt2 == '11111' then UNDEFINED;
integer t = UInt(Rt);
integer t2 = UInt(Rt2);
integer n = UInt(Rn);

boolean acquire = A == '1';
boolean release = R == '1';
boolean tagchecked = n != 31;

boolean rt_unknown = FALSE;

if t == t2 then
    Constraint c = ConstrainUnpredictable(Unpredictable_LSE128OVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
```

```
        when Constraint_UNKNOWN rt_unknown = TRUE;    // result is UNKNOWN
        when Constraint_UNDEF   UNDEFINED;
        when Constraint_NOP     EndOfInstruction();
```

Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*, and particularly *CONSTRAINED UNPREDICTABLE behavior for A64 instructions*.

Assembler symbols

<Xt1>	Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.
<Xt2>	Is the 64-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
if !IsD128Enabled(PSTATE.EL) then UNDEFINED;
bits(64) address;
bits(64) value1;
bits(64) value2;
bits(128) newdata;
bits(128) readdata;
bits(4) nzcv;

AccessDescriptor accdesc = CreateAccDescRCW(MemAtomicOp_ORR, FALSE, acquire, release, tagchecked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

value1 = X[t, 64];
value2 = X[t2, 64];

newdata = if BigEndian(accdesc.acctype) then value1:value2 else value2:value1;

bits(128) compdata = bits(128) UNKNOWN;    // Irrelevant when not executing CAS
(nzcv, readdata) = MemAtomicRCW(address, compdata, newdata, accdesc);

PSTATE.<N,Z,C,V> = nzcv;
if rt_unknown then
    readdata = bits(128) UNKNOWN;

if BigEndian(accdesc.acctype) then
    X[t, 64] = readdata<127:64>;
    X[t2, 64] = readdata<63:0>;
else
    X[t, 64] = readdata<63:0>;
    X[t2, 64] = readdata<127:64>;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

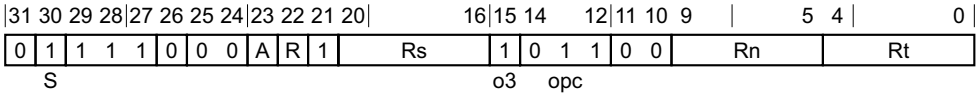
C6.2.285 RCWSSET, RCWSSETA, RCWSSETL, RCWSSETAL

Read Check Write Software atomic bit Set on doubleword in memory atomically loads a 64-bit doubleword from memory, performs a bitwise OR with the complement of the value held in a register on it, and conditionally stores the result back to memory. Storing of the result back to memory is conditional on RCW Checks and RCWS Checks. The value initially loaded from memory is returned in the destination register. This instruction updates the condition flags based on the result of the update of memory.

- RCWSSETA and RCWSSETAL load from memory with acquire semantics.
- RCWSSETL and RCWSSETAL store to memory with release semantics.
- RCWSSET has neither acquire nor release semantics.

Integer

(FEAT_THE)



RCWSSET variant

Applies when A == 0 && R == 0.

```
RCWSSET <Xs>, <Xt>, [<Xn|SP>]
```

RCWSSETA variant

Applies when A == 1 && R == 0.

```
RCWSSETA <Xs>, <Xt>, [<Xn|SP>]
```

RCWSSETAL variant

Applies when A == 1 && R == 1.

```
RCWSSETAL <Xs>, <Xt>, [<Xn|SP>]
```

RCWSSETL variant

Applies when A == 0 && R == 1.

```
RCWSSETL <Xs>, <Xt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_THE) then UNDEFINED;
integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

boolean acquire = A == '1' && Rt != '11111';
boolean release = R == '1';
boolean tagchecked = n != 31;
```

Assembler symbols

<Xs> Is the 64-bit name of the general-purpose register to be stored, encoded in the "Rs" field.

<Xt> Is the 64-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
if IsD128Enabled(PSTATE.EL) then UNDEFINED;
bits(64) address;
bits(64) newdata = X[s, 64];
bits(64) readdata;
bits(4) nzcv;

AccessDescriptor accdesc = CreateAccDescRCW(MemAtomicOp_ORR, TRUE, acquire, release, tagchecked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

bits(64) compdata = bits(64) UNKNOWN;    // Irrelevant when not executing CAS
(nzcv, readdata) = MemAtomicRCW(address, compdata, newdata, accdesc);

PSTATE.<N,Z,C,V> = nzcv;
X[t, 64] = readdata;    // Return the old value when t!=31
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

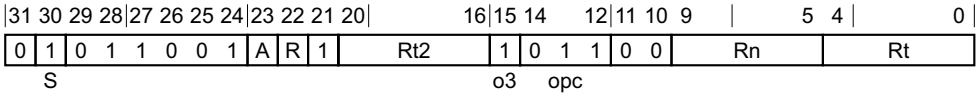
C6.2.286 RCWSSETP, RCWSSETPA, RCWSSETPL, RCWSSETPAL

Read Check Write Software atomic bit Set on quadword in memory atomically loads a 128-bit quadword from memory, performs a bitwise OR with the value held in a pair of registers on it, and conditionally stores the result back to memory. Storing of the result back to memory is conditional on RCW Checks and RCWS Checks. The value initially loaded from memory is returned in the same pair of registers. This instruction updates the condition flags based on the result of the update of memory.

- RCWSSETPA and RCWSSETPAL load from memory with acquire semantics.
- RCWSSETPL and RCWSSETPAL store to memory with release semantics.
- RCWSSETP has neither acquire nor release semantics.

Integer

(FEAT_D128 && FEAT_THE)



RCWSSETP variant

Applies when A == 0 && R == 0.

```
RCWSSETP <Xt1>, <Xt2>, [<Xn|SP>]
```

RCWSSETPA variant

Applies when A == 1 && R == 0.

```
RCWSSETPA <Xt1>, <Xt2>, [<Xn|SP>]
```

RCWSSETPAL variant

Applies when A == 1 && R == 1.

```
RCWSSETPAL <Xt1>, <Xt2>, [<Xn|SP>]
```

RCWSSETPL variant

Applies when A == 0 && R == 1.

```
RCWSSETPL <Xt1>, <Xt2>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_D128) || !IsFeatureImplemented(FEAT_THE) then UNDEFINED;
if Rt == '11111' then UNDEFINED;
if Rt2 == '11111' then UNDEFINED;
integer t = UInt(Rt);
integer t2 = UInt(Rt2);
integer n = UInt(Rn);

boolean acquire = A == '1';
boolean release = R == '1';
boolean tagchecked = n != 31;

boolean rt_unknown = FALSE;

if t == t2 then
    Constraint c = ConstrainUnpredictable(Unpredictable_LSE128OVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
```

```
when Constraint_UNKNOWN rt_unknown = TRUE;    // result is UNKNOWN
when Constraint_UNDEF   UNDEFINED;
when Constraint_NOP     EndOfInstruction();
```

Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*, and particularly *CONSTRAINED UNPREDICTABLE behavior for A64 instructions*.

Assembler symbols

<Xt1>	Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.
<Xt2>	Is the 64-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
if !IsD128Enabled(PSTATE.EL) then UNDEFINED;
bits(64) address;
bits(64) value1;
bits(64) value2;
bits(128) newdata;
bits(128) readdata;
bits(4) nzcv;

AccessDescriptor accdesc = CreateAccDescRCW(MemAtomicOp_ORR, TRUE, acquire, release, tagchecked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

value1 = X[t, 64];
value2 = X[t2, 64];

newdata = if BigEndian(accdesc.acctype) then value1:value2 else value2:value1;

bits(128) compdata = bits(128) UNKNOWN;    // Irrelevant when not executing CAS
(nzcv, readdata) = MemAtomicRCW(address, compdata, newdata, accdesc);

PSTATE.<N,Z,C,V> = nzcv;
if rt_unknown then
    readdata = bits(128) UNKNOWN;

if BigEndian(accdesc.acctype) then
    X[t, 64] = readdata<127:64>;
    X[t2, 64] = readdata<63:0>;
else
    X[t, 64] = readdata<63:0>;
    X[t2, 64] = readdata<127:64>;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

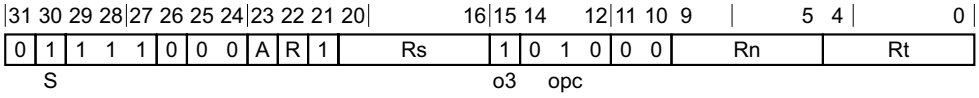
C6.2.287 RCWSSWP, RCWSSWPA, RCWSSWPL, RCWSSWPAL

Read Check Write Software Swap doubleword in memory atomically loads a 64-bit doubleword from a memory location, and conditionally stores the value held in a register back to the same memory location. Storing back to memory is conditional on RCW Checks and RCWS Checks. The value initially loaded from memory is returned in the destination register. This instruction updates the condition flags based on the result of the update of memory.

- `RCWSSWPA` and `RCWSSWPAL` load from memory with acquire semantics.
- `RCWSSWPL` and `RCWSSWPAL` store to memory with release semantics.
- `RCWSSWP` has neither acquire nor release semantics.

Integer

(FEAT_THE)



RCWSSWP variant

Applies when `A == 0 && R == 0`.

```
RCWSSWP <Xs>, <Xt>, [<Xn|SP>]
```

RCWSSWPA variant

Applies when `A == 1 && R == 0`.

```
RCWSSWPA <Xs>, <Xt>, [<Xn|SP>]
```

RCWSSWPAL variant

Applies when `A == 1 && R == 1`.

```
RCWSSWPAL <Xs>, <Xt>, [<Xn|SP>]
```

RCWSSWPL variant

Applies when `A == 0 && R == 1`.

```
RCWSSWPL <Xs>, <Xt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_THE) then UNDEFINED;
integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

boolean acquire = A == '1' && Rt != '11111';
boolean release = R == '1';
boolean tagchecked = n != 31;
```

Assembler symbols

`<Xs>`	Is the 64-bit name of the general-purpose register to be stored, encoded in the "Rs" field.
`<Xt>`	Is the 64-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
`<Xn|SP>`	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
if IsD128Enabled(PSTATE.EL) then UNDEFINED;
bits(64) address;
bits(64) newdata = X[s, 64];
bits(64) readdata;
bits(4) nzcv;

AccessDescriptor accdesc = CreateAccDescRCW(MemAtomicOp_SWP, TRUE, acquire, release, tagchecked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

bits(64) compdata = bits(64) UNKNOWN;    // Irrelevant when not executing CAS
(nzcv, readdata) = MemAtomicRCW(address, compdata, newdata, accdesc);

PSTATE.<N,Z,C,V> = nzcv;
X[t, 64] = readdata;    // Return the old value when t!=31
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

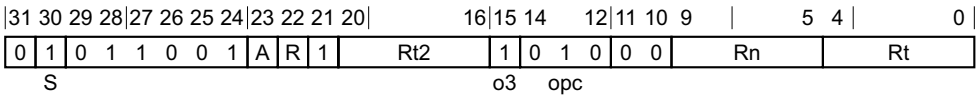
C6.2.288 RCWSSWPP, RCWSSWPPA, RCWSSWPPL, RCWSSWPPAL

Read Check Write Software Swap quadword in memory atomically loads a 128-bit quadword from a memory location, and conditionally stores the value held in a pair of registers back to the same memory location. Storing back to memory is conditional on RCW Checks and RCWS Checks. The value initially loaded from memory is returned in the same pair of registers. This instruction updates the condition flags based on the result of the update of memory.

- RCWSSWPPA and RCWSSWPPAL load from memory with acquire semantics.
- RCWSSWPPL and RCWSSWPPAL store to memory with release semantics.
- RCWSSWPP has neither acquire nor release semantics.

Integer

(FEAT_D128 && FEAT_THE)



RCWSSWPP variant

Applies when A == 0 && R == 0.

```
RCWSSWPP <Xt1>, <Xt2>, [<Xn|SP>]
```

RCWSSWPPA variant

Applies when A == 1 && R == 0.

```
RCWSSWPPA <Xt1>, <Xt2>, [<Xn|SP>]
```

RCWSSWPPAL variant

Applies when A == 1 && R == 1.

```
RCWSSWPPAL <Xt1>, <Xt2>, [<Xn|SP>]
```

RCWSSWPPL variant

Applies when A == 0 && R == 1.

```
RCWSSWPPL <Xt1>, <Xt2>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_D128) || !IsFeatureImplemented(FEAT_THE) then UNDEFINED;
if Rt == '11111' then UNDEFINED;
if Rt2 == '11111' then UNDEFINED;
integer t = UInt(Rt);
integer t2 = UInt(Rt2);
integer n = UInt(Rn);

boolean acquire = A == '1';
boolean release = R == '1';
boolean tagchecked = n != 31;

boolean rt_unknown = FALSE;

if t == t2 then
    Constraint c = ConstrainUnpredictable(Unpredictable_LSE128OVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
```

```
        when Constraint_UNKNOWN rt_unknown = TRUE;    // result is UNKNOWN
        when Constraint_UNDEF   UNDEFINED;
        when Constraint_NOP     EndOfInstruction();
```

Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*, and particularly *CONSTRAINED UNPREDICTABLE behavior for A64 instructions*.

Assembler symbols

<Xt1>	Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.
<Xt2>	Is the 64-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
if !IsD128Enabled(PSTATE.EL) then UNDEFINED;
bits(64) address;
bits(64) value1;
bits(64) value2;
bits(128) newdata;
bits(128) readdata;
bits(4) nzcv;

AccessDescriptor accdesc = CreateAccDescRCW(MemAtomicOp_SWP, TRUE, acquire, release, tagchecked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

value1 = X[t, 64];
value2 = X[t2, 64];

newdata = if BigEndian(accdesc.acctype) then value1:value2 else value2:value1;

bits(128) compdata = bits(128) UNKNOWN;    // Irrelevant when not executing CAS
(nzcv, readdata) = MemAtomicRCW(address, compdata, newdata, accdesc);

PSTATE.<N,Z,C,V> = nzcv;
if rt_unknown then
    readdata = bits(128) UNKNOWN;

if BigEndian(accdesc.acctype) then
    X[t, 64] = readdata<127:64>;
    X[t2, 64] = readdata<63:0>;
else
    X[t, 64] = readdata<63:0>;
    X[t2, 64] = readdata<127:64>;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

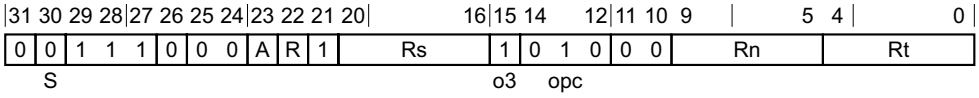
C6.2.289 RCWSWP, RCWSWPA, RCWSWPL, RCWSWPAL

Read Check Write Swap doubleword in memory atomically loads a 64-bit doubleword from a memory location, and conditionally stores the value held in a register back to the same memory location. Storing back to memory is conditional on RCW Checks. The value initially loaded from memory is returned in the destination register. This instruction updates the condition flags based on the result of the update of memory.

- `RCWSWPA` and `RCWSWPAL` load from memory with acquire semantics.
- `RCWSWPL` and `RCWSWPAL` store to memory with release semantics.
- `RCWSWP` has neither acquire nor release semantics.

Integer

(FEAT_THE)



RCWSWP variant

Applies when `A == 0 && R == 0`.

```
RCWSWP <Xs>, <Xt>, [<Xn|SP>]
```

RCWSWPA variant

Applies when `A == 1 && R == 0`.

```
RCWSWPA <Xs>, <Xt>, [<Xn|SP>]
```

RCWSWPAL variant

Applies when `A == 1 && R == 1`.

```
RCWSWPAL <Xs>, <Xt>, [<Xn|SP>]
```

RCWSWPL variant

Applies when `A == 0 && R == 1`.

```
RCWSWPL <Xs>, <Xt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_THE) then UNDEFINED;
integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

boolean acquire = A == '1' && Rt != '11111';
boolean release = R == '1';
boolean tagchecked = n != 31;
```

Assembler symbols

`<Xs>`	Is the 64-bit name of the general-purpose register to be stored, encoded in the "Rs" field.
`<Xt>`	Is the 64-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
`<Xn|SP>`	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
if IsD128Enabled(PSTATE.EL) then UNDEFINED;
bits(64) address;
bits(64) newdata = X[s, 64];
bits(64) readdata;
bits(4) nzcv;

AccessDescriptor accdesc = CreateAccDescRCW(MemAtomicOp_SWP, FALSE, acquire, release, tagchecked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

bits(64) compdata = bits(64) UNKNOWN;    // Irrelevant when not executing CAS
(nzcv, readdata) = MemAtomicRCW(address, compdata, newdata, accdesc);

PSTATE.<N,Z,C,V> = nzcv;
X[t, 64] = readdata;    // Return the old value when t!=31
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

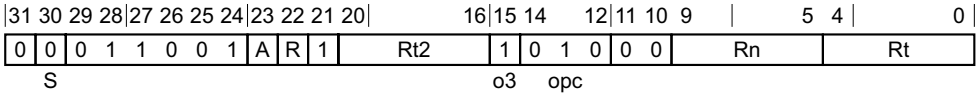
C6.2.290 RCWSWPP, RCWSWPPA, RCWSWPPL, RCWSWPPAL

Read Check Write Swap quadword in memory atomically loads a 128-bit quadword from a memory location, and conditionally stores the value held in a pair of registers back to the same memory location. Storing back to memory is conditional on RCW Checks. The value initially loaded from memory is returned in the same pair of registers. This instruction updates the condition flags based on the result of the update of memory.

- RCWSWPPA and RCWSWPPAL load from memory with acquire semantics.
- RCWSWPPL and RCWSWPPAL store to memory with release semantics.
- RCWSWPP has neither acquire nor release semantics.

Integer

(FEAT_D128 && FEAT_THE)



RCWSWPP variant

Applies when A == 0 && R == 0.

```
RCWSWPP <Xt1>, <Xt2>, [<Xn|SP>]
```

RCWSWPPA variant

Applies when A == 1 && R == 0.

```
RCWSWPPA <Xt1>, <Xt2>, [<Xn|SP>]
```

RCWSWPPAL variant

Applies when A == 1 && R == 1.

```
RCWSWPPAL <Xt1>, <Xt2>, [<Xn|SP>]
```

RCWSWPPL variant

Applies when A == 0 && R == 1.

```
RCWSWPPL <Xt1>, <Xt2>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_D128) || !IsFeatureImplemented(FEAT_THE) then UNDEFINED;
if Rt == '11111' then UNDEFINED;
if Rt2 == '11111' then UNDEFINED;
integer t = UInt(Rt);
integer t2 = UInt(Rt2);
integer n = UInt(Rn);

boolean acquire = A == '1';
boolean release = R == '1';
boolean tagchecked = n != 31;

boolean rt_unknown = FALSE;

if t == t2 then
    Constraint c = ConstrainUnpredictable(Unpredictable_LSE128OVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN rt_unknown = TRUE;    // result is UNKNOWN
```

```
        when Constraint_UNDEF   UNDEFINED;
        when Constraint_NOP     EndOfInstruction();
```

Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*, and particularly *CONSTRAINED UNPREDICTABLE behavior for A64 instructions*.

Assembler symbols

<Xt1>	Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.
<Xt2>	Is the 64-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

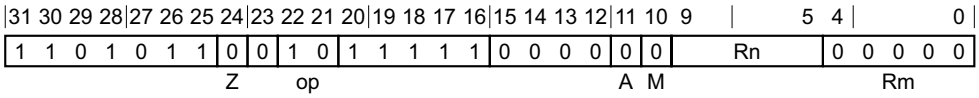
```
if !IsD128Enabled(PSTATE.EL) then UNDEFINED;
bits(64) address;
bits(64) value1;
bits(64) value2;
bits(128) newdata;
bits(128) readdata;
bits(4) nzcv;

AccessDescriptor accdesc = CreateAccDescRCW(MemAtomicOp_SWP, FALSE, acquire, release, tagchecked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

value1 = X[t, 64];
value2 = X[t2, 64];

newdata = if BigEndian(accdesc.acctype) then value1:value2 else value2:value1;

bits(128) compdata = bits(128) UNKNOWN;    // Irrelevant when not executing CAS
(nzcv, readdata) = MemAtomicRCW(address, compdata, newdata, accdesc);

PSTATE.<N,Z,C,V> = nzcv;
if rt_unknown then
    readdata = bits(128) UNKNOWN;

if BigEndian(accdesc.acctype) then
    X[t, 64] = readdata<127:64>;
    X[t2, 64] = readdata<63:0>;
else
    X[t, 64] = readdata<63:0>;
    X[t2, 64] = readdata<127:64>;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

C6.2.291 RET

Return from subroutine branches unconditionally to an address in a register, with a hint that this is a subroutine return.



Encoding

```
RET {<Xn>}
```

Decode for this encoding

```
integer n = UInt(Rn);
```

Assembler symbols

<Xn> Is the 64-bit name of the general-purpose register holding the address to be branched to, encoded in the "Rn" field. Defaults to X30 if absent.

Operation

```
bits(64) target = X[n, 64];

if (IsFeatureImplemented(FEAT_GCS) && GCSPCREnabled(PSTATE.EL)) then
    target = LoadCheckGCSRecord(target, GCSInstType_PRET);
    SetCurrentGCSPointer(GetCurrentGCSPointer() + 8);

// Value in BTypeNext will be used to set PSTATE.BTYPE
BTypeNext = '00';

BranchTo(target, BranchType_RET, FALSE);
```

C6.2.292 RETAA, RETAB

Return from subroutine, with pointer authentication. This instruction authenticates the address that is held in LR, using SP as the modifier and the specified key, and branches to the authenticated address, with a hint that this instruction is a subroutine return.

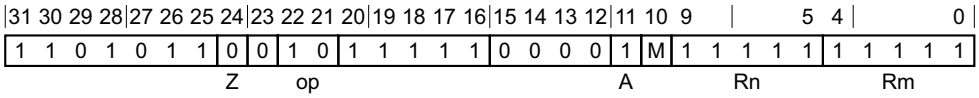
Key A is used for `RETAA`. Key B is used for `RETAB`.

If the authentication passes, the PE continues execution at the target of the branch. For information on behavior if the authentication fails, see *Faulting on pointer authentication*.

The authenticated address is not written back to LR.

Integer

(FEAT_PAuth)



RETAA variant

Applies when `M == 0`.

```
RETAA
```

RETAB variant

Applies when `M == 1`.

```
RETAB
```

Decode for all variants of this encoding

```
boolean use_key_a = (M == '0');

if !IsFeatureImplemented(FEAT_PAuth) then
    UNDEFINED;
```

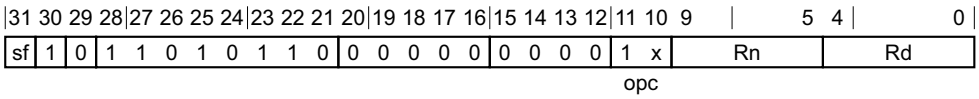
Operation

```
GCSInstruction inst_type;
bits(64) target = X[30, 64];

bits(64) modifier = SP[];

if use_key_a then
    target = AuthIA(target, modifier, TRUE);
else
    target = AuthIB(target, modifier, TRUE);

if (IsFeatureImplemented(FEAT_GCS) && GCSPCREnabled(PSTATE.EL)) then
    inst_type = if use_key_a then GCSInstType_PRETAA else GCSInstType_PRETAB;
    target = LoadCheckGCSRecord(target, inst_type);
    SetCurrentGCSPointer(GetCurrentGCSPointer() + 8);

// Value in BTypeNext will be used to set PSTATE.BTYPE
BTypeNext = '00';

BranchTo(target, BranchType_RET, FALSE);
```

C6.2.293 REV

Reverse Bytes reverses the byte order in a register.

This instruction is used by the pseudo-instruction REV64. The pseudo-instruction is never the preferred disassembly.



32-bit variant

Applies when `sf == 0 && opc == 10`.

```
REV <Wd>, <Wn>
```

64-bit variant

Applies when `sf == 1 && opc == 11`.

```
REV <Xd>, <Xn>
```

Decode for all variants of this encoding

```
if opc == '11' && sf == '0' then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

constant integer datasize = 32 << UInt(sf);
constant integer container_size = 8 << UInt(opc);
```

Assembler symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.

Operation

```
bits(datasize) operand = X[n, datasize];
bits(datasize) result;

constant integer containers = datasize DIV container_size;
for c = 0 to containers-1
    bits(container_size) container = Elem[operand, c, container_size];
    Elem[result, c, container_size] = Reverse(container, 8);

X[d, datasize] = result;
```

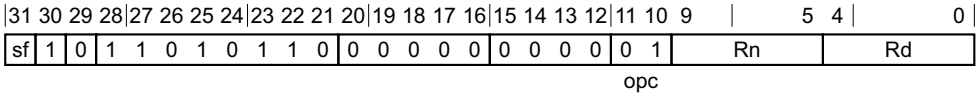
Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

C6.2.294 REV16

Reverse bytes in 16-bit halfwords reverses the byte order in each 16-bit halfword of a register.



32-bit variant

Applies when `sf == 0`.

```
REV16 <Wd>, <Wn>
```

64-bit variant

Applies when `sf == 1`.

```
REV16 <Xd>, <Xn>
```

Decode for all variants of this encoding

```
if opc == '11' && sf == '0' then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

constant integer datasize = 32 << UInt(sf);
constant integer container_size = 8 << UInt(opc);
```

Assembler symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Wn> Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Xn> Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.

Operation

```
bits(datasize) operand = X[n, datasize];
bits(datasize) result;

constant integer containers = datasize DIV container_size;
for c = 0 to containers-1
    bits(container_size) container = Elem[operand, c, container_size];
    Elem[result, c, container_size] = Reverse(container, 8);

X[d, datasize] = result;
```

Operational information

If PSTATE.DIT is 1:

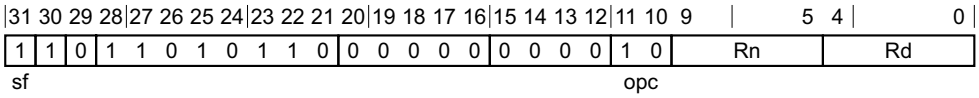
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.

  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

C6.2.295 REV32

Reverse bytes in 32-bit words reverses the byte order in each 32-bit word of a register.



Encoding

```
REV32 <Xd>, <Xn>
```

Decode for this encoding

```
if opc == '11' && sf == '0' then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

constant integer datasize = 32 << UInt(sf);
constant integer container_size = 8 << UInt(opc);
```

Assembler symbols

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Xn> Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.

Operation

```
bits(datasize) operand = X[n, datasize];
bits(datasize) result;

constant integer containers = datasize DIV container_size;
for c = 0 to containers-1
    bits(container_size) container = Elem[operand, c, container_size];
    Elem[result, c, container_size] = Reverse(container, 8);

X[d, datasize] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

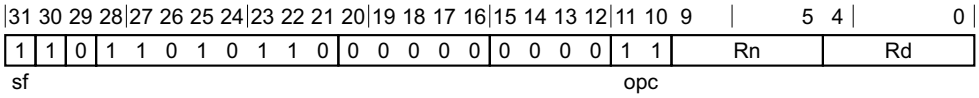
C6.2.296 REV64

Reverse Bytes reverses the byte order in a 64-bit general-purpose register.

When assembling for Armv8.2, an assembler must support this pseudo-instruction. It is OPTIONAL whether an assembler supports this pseudo-instruction when assembling for an architecture earlier than Armv8.2.

This instruction is a pseudo-instruction of the REV instruction. This means that:

- The encodings in this description are named to match the encodings of REV.
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of REV gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.



64-bit variant

```
REV64 <Xd>, <Xn>
```

is equivalent to

```
REV   <Xd>, <Xn>
```

and is never the preferred disassembly.

Assembler symbols

<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.

Operation

The description of REV gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

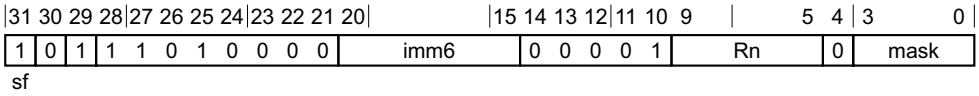
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

C6.2.297 RMIF

Performs a rotation right of a value held in a general purpose register by an immediate value, and then inserts a selection of the bottom four bits of the result of the rotation into the PSTATE flags, under the control of a second immediate mask.

Integer

(FEAT_FlagM)



Encoding

```
RMIF <Xn>, #<shift>, #<mask>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_FlagM) then UNDEFINED;
constant integer lsb = UInt(imm6);
integer n = UInt(Rn);
```

Assembler symbols

<Xn>	Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
<shift>	Is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field,
<mask>	Is the flag bit mask, an immediate in the range 0 to 15, which selects the bits that are inserted into the NZCV condition flags, encoded in the "mask" field.

Operation

```
bits(4) tmp;
bits(64) tmpreg = X[n, 64];
tmp = (tmpreg:tmpreg)<lsb+3:lsb>;
if mask<3> == '1' then PSTATE.N = tmp<3>;
if mask<2> == '1' then PSTATE.Z = tmp<2>;
if mask<1> == '1' then PSTATE.C = tmp<1>;
if mask<0> == '1' then PSTATE.V = tmp<0>;
```

Operational information

If PSTATE.DIT is 1:

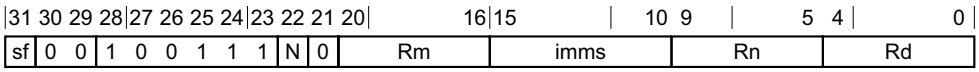
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

C6.2.298 ROR (immediate)

Rotate right (immediate) provides the value of the contents of a register rotated by a variable number of bits. The bits that are rotated off the right end are inserted into the vacated bit positions on the left.

This instruction is an alias of the EXTR instruction. This means that:

- The encodings in this description are named to match the encodings of EXTR.
- The description of EXTR gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.



32-bit variant

Applies when `sf == 0 && N == 0 && imms == 0xxxxx`.

`ROR <Wd>, <Ws>, #<shift>`

is equivalent to

`EXTR <Wd>, <Ws>, <Ws>, #<shift>`

and is the preferred disassembly when `Rn == Rm`.

64-bit variant

Applies when `sf == 1 && N == 1`.

`ROR <Xd>, <Xs>, #<shift>`

is equivalent to

`EXTR <Xd>, <Xs>, <Xs>, #<shift>`

and is the preferred disassembly when `Rn == Rm`.

Assembler symbols

`<Wd>`	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
`<Ws>`	Is the 32-bit name of the general-purpose source register, encoded in the "Rn" and "Rm" fields.
`<Xd>`	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
`<Xs>`	Is the 64-bit name of the general-purpose source register, encoded in the "Rn" and "Rm" fields.
`<shift>`	For the 32-bit variant: is the amount by which to rotate, in the range 0 to 31, encoded in the "imms" field. For the 64-bit variant: is the amount by which to rotate, in the range 0 to 63, encoded in the "imms" field.

Operation

The description of EXTR gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

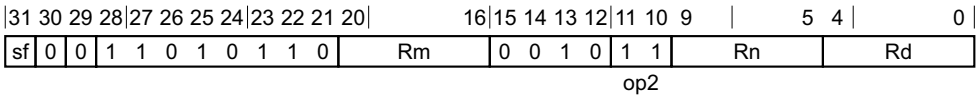
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

C6.2.299 ROR (register)

Rotate Right (register) provides the value of the contents of a register rotated by a variable number of bits. The bits that are rotated off the right end are inserted into the vacated bit positions on the left. The remainder obtained by dividing the second source register by the data size defines the number of bits by which the first source register is right-shifted.

This instruction is an alias of the RORV instruction. This means that:

- The encodings in this description are named to match the encodings of RORV.
- The description of RORV gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.



32-bit variant

Applies when `sf == 0`.

```
ROR <Wd>, <Wn>, <Wm>
```

is equivalent to

```
RORV <Wd>, <Wn>, <Wm>
```

and is always the preferred disassembly.

64-bit variant

Applies when `sf == 1`.

```
ROR <Xd>, <Xn>, <Xm>
```

is equivalent to

```
RORV <Xd>, <Xn>, <Xm>
```

and is always the preferred disassembly.

Assembler symbols

`<Wd>`	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
`<Wn>`	Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
`<Wm>`	Is the 32-bit name of the second general-purpose source register holding a shift amount from 0 to 31 in its bottom 5 bits, encoded in the "Rm" field.
`<Xd>`	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
`<Xn>`	Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
`<Xm>`	Is the 64-bit name of the second general-purpose source register holding a shift amount from 0 to 63 in its bottom 6 bits, encoded in the "Rm" field.

Operation

The description of RORV gives the operational pseudocode for this instruction.

Operational information

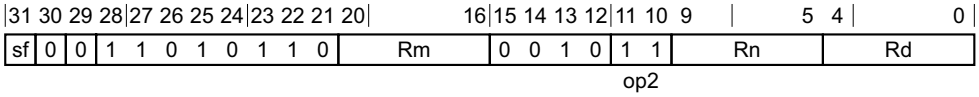
If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

C6.2.300 RORV

Rotate Right Variable provides the value of the contents of a register rotated by a variable number of bits. The bits that are rotated off the right end are inserted into the vacated bit positions on the left. The remainder obtained by dividing the second source register by the data size defines the number of bits by which the first source register is right-shifted.

This instruction is used by the alias ROR (register). The alias is always the preferred disassembly.



32-bit variant

Applies when `sf == 0`.

```
RORV <Wd>, <Wn>, <Wm>
```

64-bit variant

Applies when `sf == 1`.

```
RORV <Xd>, <Xn>, <Xm>
```

Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
constant integer datasize = 32 << UInt(sf);
ShiftType shift_type = DecodeShift(op2);
```

Assembler symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Wm>	Is the 32-bit name of the second general-purpose source register holding a shift amount from 0 to 31 in its bottom 5 bits, encoded in the "Rm" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the second general-purpose source register holding a shift amount from 0 to 63 in its bottom 6 bits, encoded in the "Rm" field.

Operation

```
bits(datasize) result;
bits(datasize) operand2 = X[m, datasize];

result = ShiftReg(n, shift_type, UInt(operand2) MOD datasize, datasize);
X[d, datasize] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

C6.2.301 RPRFM

Range Prefetch Memory signals the memory system that data memory accesses from a specified range of addresses are likely to occur in the near future. The instruction may also signal the memory system about the likelihood of data reuse of the specified range of addresses. The memory system can respond by taking actions that are expected to speed up the memory accesses when they do occur, such as prefetching locations within the specified address ranges into one or more caches. The memory system may also exploit the data reuse hints to decide whether to retain the data in other caches upon eviction from the innermost caches or to discard it.

The effect of an RPRFM instruction is IMPLEMENTATION DEFINED, but because these signals are only hints, the instruction cannot cause a synchronous Data Abort exception and is guaranteed not to access Device memory. It is valid for the PE to treat this instruction as a NOP.

An RPRFM instruction specifies the type of accesses and range of addresses using the following parameters:

- 'Type', in the <rprfop> operand opcode bits, specifies whether the prefetched data will be accessed by load or store instructions.
- 'Policy', in the <rprfop> operand opcode bits, specifies whether the data is likely to be reused or if it is a streaming, non-temporal prefetch. If a streaming prefetch is specified, then the 'ReuseDistance' parameter is ignored.
- 'BaseAddress', in the 64-bit base register, holds the initial block address for the accesses.
- 'ReuseDistance', in the metadata register bits[63:60], indicates the maximum number of bytes to be accessed by this PE before executing the next RPRFM instruction that specifies the same range. This includes the total number of bytes inside and outside of the range that will be accessed by the same PE. This parameter can be used to influence cache eviction and replacement policies, in order to retain the data in the most optimal levels of the memory hierarchy after each access. If software cannot easily determine the amount of other memory that will be accessed, these bits can be set to zero to indicate that 'ReuseDistance' is not known. Otherwise, these four bits encode decreasing powers of two in the range 512MiB (0b0001) to 32KiB (0b1111).
- 'Stride', in the metadata register bits[59:38], is a signed, two's complement integer encoding of the number of bytes to advance the block address after 'Length' bytes have been accessed, in the range -2MiB to +2MiB-1B. A negative value indicates that the block address is advanced in a descending direction.
- 'Count', in the metadata register bits[37:22], is an unsigned integer encoding of the number of blocks of data to be accessed minus 1, representing the range 1 to 65536 blocks. If 'Count' is 0, then the 'Stride' parameter is ignored and only a single block of contiguous bytes from 'BaseAddress' to ('BaseAddress' + 'Length' - 1) is described.
- 'Length', in the metadata register bits[21:0], is a signed, two's complement integer encoding of the number of contiguous bytes to be accessed starting from the current block address, without changing the block address, in the range -2MiB to +2MiB-1B. A negative value indicates that the bytes are accessed in a descending direction.

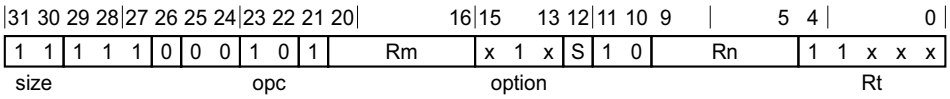
Note

Software is expected to honor the parameters it provides to the RPRFM instruction, and the same PE should access all locations in the range, in the direction specified by the sign of the 'Length' and 'Stride' parameters. A range prefetch is considered active on a PE until all locations in the range have been accessed by the PE. A range prefetch might also be inactivated by the PE prior to completion, for example due to a software context switch or lack of hardware resources.

Software should not specify overlapping addresses in multiple active ranges. If a range is expected to be accessed by both load and store instructions (read-modify-write), then a single range with a 'Type' parameter of PST (prefetch for store) should be specified.

Integer

(FEAT_RPRFM)



Encoding

```
RPRFM (<rprfop>|#<imm6>), <Xm>, [<Xn|SP>]
```

Decode for this encoding

```
bits(6) operation = option<2>:option<0>:S:Rt<2:0>;
integer n = UInt(Rn);
integer m = UInt(Rm);
```

Assembler symbols

`<rprfop>` Is the range prefetch operation, defined as `<type><policy>`.

`<type>` is one of:

`PLD` Prefetch for load, encoded in the "Rt<0>" field as `0`.

`PST` Prefetch for store, encoded in the "Rt<0>" field as `1`.

`<policy>` is one of:

`KEEP` Retained or temporal prefetch, for data that is expected to be kept in caches to be accessed more than once, encoded in the "option<2>:option<0>:S:Rt<2:1>" fields as `0b00000`.

`STRM` Streaming or non-temporal prefetch, for data that is expected to be accessed once and not reused, encoded in the "option<2>:option<0>:S:Rt<2:1>" fields as `0b00010`.

For other encodings of the "option<2>:option<0>:S:Rt<2:0>" fields, use `<imm6>`.

`<imm6>` Is the range prefetch operation encoding as an immediate, in the range 0 to 63, encoded in "option<2>:option<0>:S:Rt<2:0>". This syntax is only for encodings that are not representable using `<rprfop>`.

`<Xm>` Is the 64-bit name of the general-purpose register that holds an encoding of the metadata, encoded in the "Rm" field.

`<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

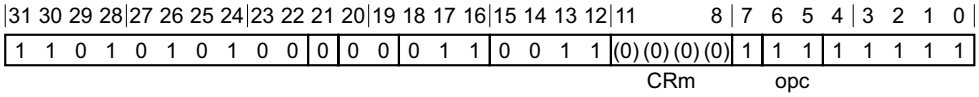
```
bits(64) address = if n == 31 then SP[] else X[n, 64];
bits(64) metadata = X[m, 64];
integer stride = SInt(metadata<59:38>);
integer count = UInt(metadata<37:22>) + 1;
integer length = SInt(metadata<21:0>);
integer reuse;

if metadata<63:60> == '0000' then
    reuse = -1;    // Not known
else
    reuse = 32768 << (15 - UInt(metadata<63:60>));

Hint_RangePrefetch(address, length, stride, count, reuse, operation);
```

C6.2.302 SB

Speculation Barrier is a barrier that controls speculation. For more information and details of the semantics, see *Speculation Barrier (SB)*.

System

(FEAT_SB)



Encoding

SB

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SB) then UNDEFINED;
```

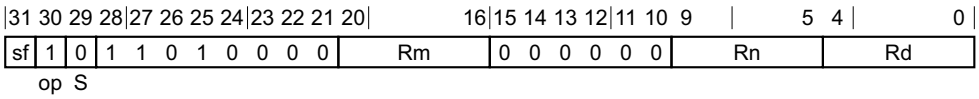
Operation

```
SpeculationBarrier();
```

C6.2.303 SBC

Subtract with Carry subtracts a register value and the value of NOT (Carry flag) from a register value, and writes the result to the destination register.

This instruction is used by the alias NGC. See *Alias conditions* for details of when each alias is preferred.



32-bit variant

Applies when `sf == 0`.

```
SBC <Wd>, <Wn>, <Wm>
```

64-bit variant

Applies when `sf == 1`.

```
SBC <Xd>, <Xn>, <Xm>
```

Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
constant integer datasize = 32 << UInt(sf);
```

Alias conditions

Alias	is preferred when
NGC	`Rn == '11111'`

Assembler symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Wm>	Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.

Operation

```
bits(datasize) result;
bits(datasize) operand1 = X[n, datasize];
bits(datasize) operand2 = X[m, datasize];

operand2 = NOT(operand2);
```

```
(result, -) = AddWithCarry(operand1, operand2, PSTATE.C);

X[d, datasize] = result;
```

Operational information

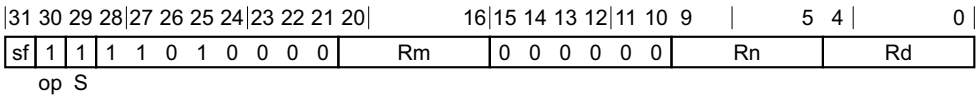
If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

C6.2.304 SBCS

Subtract with Carry, setting flags, subtracts a register value and the value of NOT (Carry flag) from a register value, and writes the result to the destination register. It updates the condition flags based on the result.

This instruction is used by the alias NGCS. See *Alias conditions* for details of when each alias is preferred.



32-bit variant

Applies when `sf == 0`.

```
SBCS <Wd>, <Wn>, <Wm>
```

64-bit variant

Applies when `sf == 1`.

```
SBCS <Xd>, <Xn>, <Xm>
```

Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
constant integer datasize = 32 << UInt(sf);
```

Alias conditions

Alias	is preferred when
NGCS	`Rn == '11111'`

Assembler symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Wm>	Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.

Operation

```
bits(datasize) result;
bits(datasize) operand1 = X[n, datasize];
bits(datasize) operand2 = X[m, datasize];
bits(4) nzcv;

operand2 = NOT(operand2);
```

```
(result, nzcv) = AddWithCarry(operand1, operand2, PSTATE.C);

PSTATE.<N,Z,C,V> = nzcv;

X[d, datasize] = result;
```

Operational information

If PSTATE.DIT is 1:

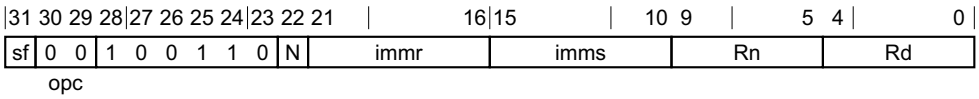
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

C6.2.305 SBFIZ

Signed Bitfield Insert in Zeros copies a bitfield of <width> bits from the least significant bits of the source register to bit position <lsb> of the destination register, setting the destination bits below the bitfield to zero, and the bits above the bitfield to a copy of the most significant bit of the bitfield.

This instruction is an alias of the SBFM instruction. This means that:

- The encodings in this description are named to match the encodings of SBFM.
- The description of SBFM gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.



32-bit variant

Applies when sf == 0 && N == 0.

```
SBFIZ <Wd>, <Wn>, #<lsb>, #<width>
```

is equivalent to

```
SBFM <Wd>, <Wn>, #(-<lsb> MOD 32), #(<width>-1)
```

and is the preferred disassembly when UInt(imms) < UInt(immr).

64-bit variant

Applies when sf == 1 && N == 1.

```
SBFIZ <Xd>, <Xn>, #<lsb>, #<width>
```

is equivalent to

```
SBFM <Xd>, <Xn>, #(-<lsb> MOD 64), #(<width>-1)
```

and is the preferred disassembly when UInt(imms) < UInt(immr).

Assembler symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
<lsb>	For the 32-bit variant: is the bit number of the lsb of the destination bitfield, in the range 0 to 31. For the 64-bit variant: is the bit number of the lsb of the destination bitfield, in the range 0 to 63.
<width>	For the 32-bit variant: is the width of the bitfield, in the range 1 to 32-<lsb>. For the 64-bit variant: is the width of the bitfield, in the range 1 to 64-<lsb>.

Operation

The description of SBFM gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

C6.2.306 SBFM

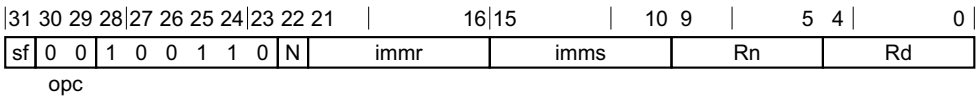
Signed Bitfield Move is usually accessed via one of its aliases, which are always preferred for disassembly.

If <imms> is greater than or equal to <immr>, this copies a bitfield of (<imms>-<immr>+1) bits starting from bit position <immr> in the source register to the least significant bits of the destination register.

If <imms> is less than <immr>, this copies a bitfield of (<imms>+1) bits from the least significant bits of the source register to bit position (regsize-<immr>) of the destination register, where regsize is the destination register size of 32 or 64 bits.

In both cases the destination bits below the bitfield are set to zero, and the bits above the bitfield are set to a copy of the most significant bit of the bitfield.

This instruction is used by the aliases ASR (immediate), SBFIZ, SBFX, SXTB, SXTH, and SXTW. See *Alias conditions* for details of when each alias is preferred.



32-bit variant

Applies when sf == 0 && N == 0.

```
SBFM <Wd>, <Wn>, #<immr>, #<imms>
```

64-bit variant

Applies when sf == 1 && N == 1.

```
SBFM <Xd>, <Xn>, #<immr>, #<imms>
```

Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
constant integer datasize = 32 << UInt(sf);

integer r;
integer s;
bits(datasize) wmask;
bits(datasize) tmask;

if sf == '1' && N != '1' then UNDEFINED;
if sf == '0' && (N != '0' || immr<5> != '0' || imms<5> != '0') then UNDEFINED;

r = UInt(immr);
s = UInt(imms);
(wmask, tmask) = DecodeBitMasks(N, imms, immr, FALSE, datasize);
```

Alias conditions

Alias	of variant	is preferred when
ASR (immediate)	32-bit	`imms == '011111'`
ASR (immediate)	64-bit	`imms == '111111'`
SBFIZ	-	`UInt(imms) < UInt(immr)`
SBFX	-	`BFXPreferred(sf, opc<1>, imms, immr)`
SXTB	-	`immr == '000000' && imms == '000111'`
SXTH	-	`immr == '000000' && imms == '001111'`
SXTW	-	`immr == '000000' && imms == '011111'`

Assembler symbols

`<Wd>` Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

`<Wn>` Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.

`<Xd>` Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

`<Xn>` Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.

`<immr>` For the 32-bit variant: is the right rotate amount, in the range 0 to 31, encoded in the "immr" field.

For the 64-bit variant: is the right rotate amount, in the range 0 to 63, encoded in the "immr" field.

`<imms>` For the 32-bit variant: is the leftmost bit number to be moved from the source, in the range 0 to 31, encoded in the "imms" field.

For the 64-bit variant: is the leftmost bit number to be moved from the source, in the range 0 to 63, encoded in the "imms" field.

Operation

```
bits(datasize) src = X[n, datasize];

// perform bitfield move on low bits
bits(datasize) bot = ROR(src, r) AND wmask;

// determine extension bits (sign, zero or dest register)
bits(datasize) top = Replicate(src<s>, datasize);

// combine extension bits and result bits
X[d, datasize] = (top AND NOT(tmask)) OR (bot AND tmask);
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

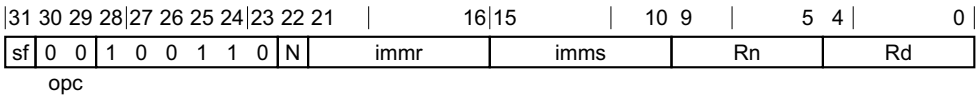
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

C6.2.307 SBFX

Signed Bitfield Extract copies a bitfield of <width> bits starting from bit position <lsb> in the source register to the least significant bits of the destination register, and sets destination bits above the bitfield to a copy of the most significant bit of the bitfield.

This instruction is an alias of the SBFM instruction. This means that:

- The encodings in this description are named to match the encodings of SBFM.
- The description of SBFM gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.



32-bit variant

Applies when `sf == 0 && N == 0`.

```
SBFX <Wd>, <Wn>, #<lsb>, #<width>
```

is equivalent to

```
SBFM <Wd>, <Wn>, #<lsb>, #(<lsb>+<width>-1)
```

and is the preferred disassembly when `BFXPreferred(sf, opc<1>, imms, immr)`.

64-bit variant

Applies when `sf == 1 && N == 1`.

```
SBFX <Xd>, <Xn>, #<lsb>, #<width>
```

is equivalent to

```
SBFM <Xd>, <Xn>, #<lsb>, #(<lsb>+<width>-1)
```

and is the preferred disassembly when `BFXPreferred(sf, opc<1>, imms, immr)`.

Assembler symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
<lsb>	For the 32-bit variant: is the bit number of the lsb of the source bitfield, in the range 0 to 31. For the 64-bit variant: is the bit number of the lsb of the source bitfield, in the range 0 to 63.
<width>	For the 32-bit variant: is the width of the bitfield, in the range 1 to 32-<lsb>. For the 64-bit variant: is the width of the bitfield, in the range 1 to 64-<lsb>.

Operation

The description of SBFM gives the operational pseudocode for this instruction.

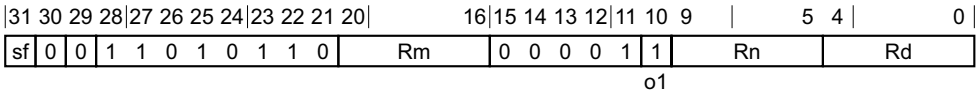
Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

C6.2.308 SDIV

Signed Divide divides a signed integer register value by another signed integer register value, and writes the result to the destination register. The condition flags are not affected.



32-bit variant

Applies when `sf == 0`.

```
SDIV <Wd>, <Wn>, <Wm>
```

64-bit variant

Applies when `sf == 1`.

```
SDIV <Xd>, <Xn>, <Xm>
```

Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
constant integer datasize = 32 << UInt(sf);
```

Assembler symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Wm>	Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.

Operation

```
bits(datasize) operand1 = X[n, datasize];
bits(datasize) operand2 = X[m, datasize];
integer result;

if IsZero(operand2) then
    result = 0;
else
    result = RoundTowardsZero(Real(Int(operand1, FALSE)) / Real(Int(operand2, FALSE)));

X[d, datasize] = result<datasize-1:0>;
```

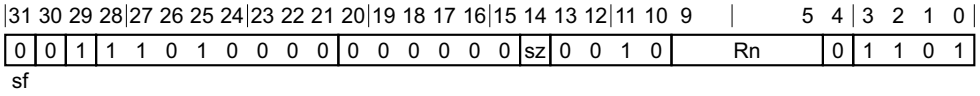
C6.2.309 SETF8, SETF16

Set the PSTATE.NZV flags based on the value in the specified general-purpose register. SETF8 treats the value as an 8 bit value, and SETF16 treats the value as an 16 bit value.

The PSTATE.C flag is not affected by these instructions.

Integer

(FEAT_FlagM)



SETF8 variant

Applies when `sz == 0`.

```
SETF8 <Wn>
```

SETF16 variant

Applies when `sz == 1`.

```
SETF16 <Wn>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_FlagM) then UNDEFINED;
constant integer msb = (8 << UInt(sz)) - 1;
integer n = UInt(Rn);
```

Assembler symbols

<Wn>	Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.

Operation

```
bits(32) tmpreg = X[n, 32];
PSTATE.N = tmpreg<msb>;
PSTATE.Z = if (tmpreg<msb:0> == Zeros(msb + 1)) then '1' else '0';
PSTATE.V = tmpreg<msb+1> EOR tmpreg<msb>;
//PSTATE.C unchanged;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

C6.2.310 SETGP, SETGM, SETGE

Memory Set with tag setting. These instructions perform a memory set using the value in the bottom byte of the source register and store an Allocation Tag to memory for each Tag Granule written. The Allocation Tag is calculated from the Logical Address Tag in the register which holds the first address that the set is made to. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: SETGP, then SETGM, and then SETGE.

SETGP performs some preconditioning of the arguments suitable for using the SETGM instruction, and performs an IMPLEMENTATION DEFINED amount of the memory set. SETGM performs an IMPLEMENTATION DEFINED amount of the memory set. SETGE performs the last part of the memory set.

Note

The inclusion of IMPLEMENTATION DEFINED amounts of memory set allows some optimization of the size that can be performed.

The architecture supports two algorithms for the memory set: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

After execution of SETGP, option A (which results in encoding PSTATE.C = 0):

- If Xn<63> == 1, the set size is saturated to `0x7FFFFFFFFFFFFF0`.
- Xd holds the original Xd + saturated Xn.
- Xn holds -1* saturated Xn + an IMPLEMENTATION DEFINED number of bytes set.
- PSTATE.{N,Z,V} are set to {0,0,0}.

After execution of SETGP, option B (which results in encoding PSTATE.C = 1):

- If Xn<63> == 1, the copy size is saturated to `0x7FFFFFFFFFFFFF0`.
- Xd holds the original Xd + an IMPLEMENTATION DEFINED number of bytes set.
- Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes set.
- PSTATE.{N,Z,V} are set to {0,0,0}.

For SETGM, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number.
- Xn holds -1* number of bytes remaining to be set in the memory set in total.
- Xd holds the lowest address that the set is made to -Xn.
- At the end of the instruction, the value of Xn is written back with -1* number of bytes remaining to be set in the memory set in total.

For SETGM, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes remaining to be set in the memory set in total.
- Xd holds the lowest address that the set is made to.
- At the end of the instruction:
  - the value of Xn is written back with the number of bytes remaining to be set in the memory set in total.
  - the value of Xd is written back with the lowest address that has not been set.

For SETGE, option A (encoded by PSTATE.C = 0), the format of the arguments is:

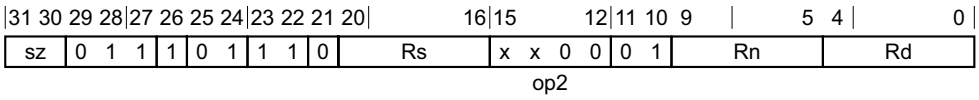
- Xn is treated as a signed 64-bit number.
- Xn holds -1* the number of bytes remaining to be set in the memory set in total.
- Xd holds the lowest address that the set is made to -Xn.
- At the end of the instruction, the value of Xn is written back with 0.

For SETGE, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes remaining to be set in the memory set in total.
- Xd holds the lowest address that the set is made to.
- At the end of the instruction:
  - the value of Xn is written back with 0.
  - the value of Xd is written back with the lowest address that has not been set.

Integer

(FEAT_MOPS)



Epilogue variant

Applies when op2 == 1000.

```
SETGE [<Xd>]!, <Xn>!, <Xs>
```

Main variant

Applies when op2 == 0100.

```
SETGM [<Xd>]!, <Xn>!, <Xs>
```

Prologue variant

Applies when op2 == 0000.

```
SETGP [<Xd>]!, <Xn>!, <Xs>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_MOPS) || !IsFeatureImplemented(FEAT_MTE) || sz != '00' then UNDEFINED;

SETParams memset;
memset.d = UInt(Rd);
memset.s = UInt(Rs);
memset.n = UInt(Rn);
bits(2) options = op2<1:0>;
boolean nontemporal = options<1> == '1';

case op2<3:2> of
    when '00' memset.stage = MOPSStage_Prologue;
    when '01' memset.stage = MOPSStage_Main;
    when '10' memset.stage = MOPSStage_Epilogue;
    otherwise UNDEFINED;

CheckMOPSEnabled();
```

```
 if (memset.s == memset.n || memset.s == memset.d || memset.n == memset.d || memset.d == 31 || memset.n
== 31) then
     Constraint c = ConstrainUnpredictable(Unpredictable_MOPSOVERLAP31);
     assert c IN {Constraint_UNDEF, Constraint_NOP};
     case c of
         when Constraint_UNDEF UNDEFINED;
         when Constraint_NOP   EndOfInstruction();
```

Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*, and particularly *Memory Copy and Memory Set SET**.

Assembler symbols

<Xd> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address (an integer multiple of 16) and for option B is updated by the instruction, encoded in the "Rd" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address (an integer multiple of 16) and is updated by the instruction, encoded in the "Rd" field.

<Xn> For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be set (an integer multiple of 16) and is set to zero at the end of the instruction, encoded in the "Rn" field.

For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be set (an integer multiple of 16) and is updated by the instruction, encoded in the "Rn" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be set (an integer multiple of 16) and is updated by the instruction, encoded in the "Rn" field.

<Xs> For the epilogue variant: is the 64-bit name of the general-purpose register that holds the source data, encoded in the "Rs" field.

For the main and prologue variant: is the 64-bit name of the general-purpose register that holds the source data in bits<7:0>, encoded in the "Rs" field.

Operation

```
bits(8) data = X[memset.s, 8];
integer B;

memset.is_setg = TRUE;
memset.nzcv = PSTATE.<N,Z,C,V>;
memset.toaddress = X[memset.d, 64];
if memset.stage == MOPSStage_Prologue then
    memset.setsize = UInt(X[memset.n, 64]);
else
    memset.setsize = SInt(X[memset.n, 64]);
memset.implements_option_a = SETGOptionA();

boolean privileged = if options<0> == '1' then AArch64.IsUnprivAccessPriv() else PSTATE.EL != EL0;

AccessDescriptor accdesc = CreateAccDescSTGMOPS(privileged, nontemporal);

if memset.stage == MOPSStage_Prologue then
    if memset.setsize > 0x7FFFFFFFFFFFFFF0 then
        memset.setsize = 0x7FFFFFFFFFFFFFF0;
```

```
    if ((memset.setsize != 0 && !IsAligned(memset.toaddress, TAG_GRANULE)) ||
!IsAligned(memset.setsize<63:0>, TAG_GRANULE)) then
        AArch64.Abort(memset.toaddress, AlignmentFault(accdesc));

    if memset.implements_option_a then
        memset.nzcv = '0000';
        memset.toaddress = memset.toaddress + memset.setsize;
        memset.setsize = 0 - memset.setsize;
    else
        memset.nzcv = '0010';

memset.stagesetsize = MemSetStageSize(memset);

if memset.stage != MOPSStage_Prologue then
    CheckMemSetParams(memset, options);

    if ((memset.setsize != 0 && !IsAligned(memset.toaddress, TAG_GRANULE)) ||
!IsAligned(memset.setsize<63:0>, TAG_GRANULE)) then
        AArch64.Abort(memset.toaddress, AlignmentFault(accdesc));

integer tagstep;
bits(4) tag;
bits(64) tagaddr;
AddressDescriptor memaddrdesc;
PhysMemRetStatus memstatus;
integer memory_set;
boolean fault = FALSE;

if memset.implements_option_a then
    while memset.stagesetsize < 0 && !fault do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = SETSizeChoice(memset, 16);
        assert B <= -1 * memset.stagesetsize && B<3:0> == '0000';

        (memory_set, memaddrdesc, memstatus) = MemSetBytes(memset.toaddress + memset.setsize, data, B,
accdesc);

        if memory_set != B then
            fault = TRUE;
        else
            tagstep = B DIV 16;
            tag = AArch64.AllocationTagFromAddress(memset.toaddress + memset.setsize);

            while tagstep > 0 do
                tagaddr = memset.toaddress + memset.setsize + (tagstep - 1) * 16;
                AArch64.MemTag[tagaddr, accdesc] = tag;
                tagstep = tagstep - 1;

            memset.setsize = memset.setsize + B;
            memset.stagesetsize = memset.stagesetsize + B;

else
    while memset.stagesetsize > 0 && !fault do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = SETSizeChoice(memset, 16);
        assert B <= memset.stagesetsize && B<3:0> == '0000';

        (memory_set, memaddrdesc, memstatus) = MemSetBytes(memset.toaddress, data, B, accdesc);

        if memory_set != B then
            fault = TRUE;
        else
            tagstep = B DIV 16;
            tag = AArch64.AllocationTagFromAddress(memset.toaddress);
            while tagstep > 0 do
                tagaddr = memset.toaddress + (tagstep - 1) * 16;
```

```
                AArch64.MemTag[tagaddr, accdesc] = tag;
                tagstep = tagstep - 1;

            memset.toaddress = memset.toaddress + B;
            memset.setsize = memset.setsize - B;
            memset.stagesetsize = memset.stagesetsize - B;

UpdateSetRegisters(memset, fault, memory_set);

if fault then
    if IsFault(memaddrdesc) then
        AArch64.Abort(memaddrdesc.vaddress, memaddrdesc.fault);
    else
        boolean iswrite = TRUE;
        HandleExternalAbort(memstatus, iswrite, memaddrdesc, B, accdesc);

if memset.stage == MOPSStage_Prologue then
    PSTATE.<N,Z,C,V> = memset.nzcv;
```

C6.2.311 SETGPN, SETGMN, SETGEN

Memory Set with tag setting, non-temporal. These instructions perform a memory set using the value in the bottom byte of the source register and store an Allocation Tag to memory for each Tag Granule written. The Allocation Tag is calculated from the Logical Address Tag in the register which holds the first address that the set is made to. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: SETGPN, then SETGMN, and then SETGEN.

SETGPN performs some preconditioning of the arguments suitable for using the SETGMN instruction, and performs an IMPLEMENTATION DEFINED amount of the memory set. SETGMN performs an IMPLEMENTATION DEFINED amount of the memory set. SETGEN performs the last part of the memory set.

Note

The inclusion of IMPLEMENTATION DEFINED amounts of memory set allows some optimization of the size that can be performed.

The architecture supports two algorithms for the memory set: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

After execution of SETGPN, option A (which results in encoding PSTATE.C = 0):

- If Xn<63> == 1, the set size is saturated to `0x7FFFFFFFFFFFFFF0`.
- Xd holds the original Xd + saturated Xn.
- Xn holds -1* saturated Xn + an IMPLEMENTATION DEFINED number of bytes set.
- PSTATE.{N,Z,V} are set to {0,0,0}.

After execution of SETGPN, option B (which results in encoding PSTATE.C = 1):

- If Xn<63> == 1, the copy size is saturated to `0x7FFFFFFFFFFFFFF0`.
- Xd holds the original Xd + an IMPLEMENTATION DEFINED number of bytes set.
- Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes set.
- PSTATE.{N,Z,V} are set to {0,0,0}.

For SETGMN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number.
- Xn holds -1* number of bytes remaining to be set in the memory set in total.
- Xd holds the lowest address that the set is made to -Xn.
- At the end of the instruction, the value of Xn is written back with -1* number of bytes remaining to be set in the memory set in total.

For SETGMN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes remaining to be set in the memory set in total.
- Xd holds the lowest address that the set is made to.
- At the end of the instruction:
  - the value of Xn is written back with the number of bytes remaining to be set in the memory set in total.
  - the value of Xd is written back with the lowest address that has not been set.

For SETGEN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

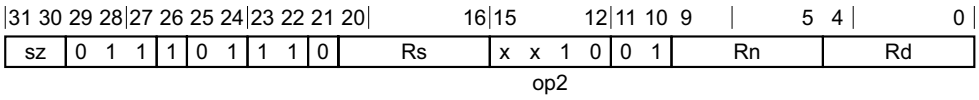
- Xn is treated as a signed 64-bit number.
- Xn holds -1* the number of bytes remaining to be set in the memory set in total.
- Xd holds the lowest address that the set is made to -Xn.
- At the end of the instruction, the value of Xn is written back with 0.

For SETGEN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes remaining to be set in the memory set in total.
- Xd holds the lowest address that the set is made to.
- At the end of the instruction:
  - the value of Xn is written back with 0.
  - the value of Xd is written back with the lowest address that has not been set.

Integer

(FEAT_MOPS)



Epilogue variant

Applies when op2 == 1010.

```
SETGEN [<Xd>]!, <Xn>!, <Xs>
```

Main variant

Applies when op2 == 0110.

```
SETGMN [<Xd>]!, <Xn>!, <Xs>
```

Prologue variant

Applies when op2 == 0010.

```
SETGPN [<Xd>]!, <Xn>!, <Xs>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_MOPS) || !IsFeatureImplemented(FEAT_MTE) || sz != '00' then UNDEFINED;

SETParams memset;
memset.d = UInt(Rd);
memset.s = UInt(Rs);
memset.n = UInt(Rn);
bits(2) options = op2<1:0>;
boolean nontemporal = options<1> == '1';

case op2<3:2> of
    when '00' memset.stage = MOPSStage_Prologue;
    when '01' memset.stage = MOPSStage_Main;
    when '10' memset.stage = MOPSStage_Epilogue;
    otherwise UNDEFINED;

CheckMOPSEnabled();
```

```
 if (memset.s == memset.n || memset.s == memset.d || memset.n == memset.d || memset.d == 31 || memset.n
== 31) then
     Constraint c = ConstrainUnpredictable(Unpredictable_MOPSOVERLAP31);
     assert c IN {Constraint_UNDEF, Constraint_NOP};
     case c of
         when Constraint_UNDEF UNDEFINED;
         when Constraint_NOP   EndOfInstruction();
```

Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*, and particularly *Memory Copy and Memory Set SET**.

Assembler symbols

<Xd> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address (an integer multiple of 16) and for option B is updated by the instruction, encoded in the "Rd" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address (an integer multiple of 16) and is updated by the instruction, encoded in the "Rd" field.

<Xn> For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be set (an integer multiple of 16) and is set to zero at the end of the instruction, encoded in the "Rn" field.

For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be set (an integer multiple of 16) and is updated by the instruction, encoded in the "Rn" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be set (an integer multiple of 16) and is updated by the instruction, encoded in the "Rn" field.

<Xs> For the epilogue variant: is the 64-bit name of the general-purpose register that holds the source data, encoded in the "Rs" field.

For the main and prologue variant: is the 64-bit name of the general-purpose register that holds the source data in bits<7:0>, encoded in the "Rs" field.

Operation

```
bits(8) data = X[memset.s, 8];
integer B;

memset.is_setg = TRUE;
memset.nzcv = PSTATE.<N,Z,C,V>;
memset.toaddress = X[memset.d, 64];
if memset.stage == MOPSStage_Prologue then
    memset.setsize = UInt(X[memset.n, 64]);
else
    memset.setsize = SInt(X[memset.n, 64]);
memset.implements_option_a = SETGOptionA();

boolean privileged = if options<0> == '1' then AArch64.IsUnprivAccessPriv() else PSTATE.EL != EL0;

AccessDescriptor accdesc = CreateAccDescSTGMOPS(privileged, nontemporal);

if memset.stage == MOPSStage_Prologue then
    if memset.setsize > 0x7FFFFFFFFFFFFFF0 then
        memset.setsize = 0x7FFFFFFFFFFFFFF0;
```

```
    if ((memset.setsize != 0 && !IsAligned(memset.toaddress, TAG_GRANULE)) ||
!IsAligned(memset.setsize<63:0>, TAG_GRANULE)) then
        AArch64.Abort(memset.toaddress, AlignmentFault(accdesc));

    if memset.implements_option_a then
        memset.nzcv = '0000';
        memset.toaddress = memset.toaddress + memset.setsize;
        memset.setsize = 0 - memset.setsize;
    else
        memset.nzcv = '0010';

memset.stagesetsize = MemSetStageSize(memset);

if memset.stage != MOPSStage_Prologue then
    CheckMemSetParams(memset, options);

    if ((memset.setsize != 0 && !IsAligned(memset.toaddress, TAG_GRANULE)) ||
!IsAligned(memset.setsize<63:0>, TAG_GRANULE)) then
        AArch64.Abort(memset.toaddress, AlignmentFault(accdesc));

integer tagstep;
bits(4) tag;
bits(64) tagaddr;
AddressDescriptor memaddrdesc;
PhysMemRetStatus memstatus;
integer memory_set;
boolean fault = FALSE;

if memset.implements_option_a then
    while memset.stagesetsize < 0 && !fault do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = SETSizeChoice(memset, 16);
        assert B <= -1 * memset.stagesetsize && B<3:0> == '0000';

        (memory_set, memaddrdesc, memstatus) = MemSetBytes(memset.toaddress + memset.setsize, data, B,
accdesc);

        if memory_set != B then
            fault = TRUE;
        else
            tagstep = B DIV 16;
            tag = AArch64.AllocationTagFromAddress(memset.toaddress + memset.setsize);

            while tagstep > 0 do
                tagaddr = memset.toaddress + memset.setsize + (tagstep - 1) * 16;
                AArch64.MemTag[tagaddr, accdesc] = tag;
                tagstep = tagstep - 1;

            memset.setsize = memset.setsize + B;
            memset.stagesetsize = memset.stagesetsize + B;

else
    while memset.stagesetsize > 0 && !fault do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = SETSizeChoice(memset, 16);
        assert B <= memset.stagesetsize && B<3:0> == '0000';

        (memory_set, memaddrdesc, memstatus) = MemSetBytes(memset.toaddress, data, B, accdesc);

        if memory_set != B then
            fault = TRUE;
        else
            tagstep = B DIV 16;
            tag = AArch64.AllocationTagFromAddress(memset.toaddress);
            while tagstep > 0 do
                tagaddr = memset.toaddress + (tagstep - 1) * 16;
```

```
                AArch64.MemTag[tagaddr, accdesc] = tag;
                tagstep = tagstep - 1;

            memset.toaddress = memset.toaddress + B;
            memset.setsize = memset.setsize - B;
            memset.stagesetsize = memset.stagesetsize - B;

UpdateSetRegisters(memset, fault, memory_set);

if fault then
    if IsFault(memaddrdesc) then
        AArch64.Abort(memaddrdesc.vaddress, memaddrdesc.fault);
    else
        boolean iswrite = TRUE;
        HandleExternalAbort(memstatus, iswrite, memaddrdesc, B, accdesc);

if memset.stage == MOPSStage_Prologue then
    PSTATE.<N,Z,C,V> = memset.nzcv;
```

C6.2.312 SETGPT, SETGMT, SETGET

Memory Set with tag setting, unprivileged. These instructions perform a memory set using the value in the bottom byte of the source register and store an Allocation Tag to memory for each Tag Granule written. The Allocation Tag is calculated from the Logical Address Tag in the register which holds the first address that the set is made to. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: SETGPT, then SETGMT, and then SETGET.

SETGPT performs some preconditioning of the arguments suitable for using the SETGMT instruction, and performs an IMPLEMENTATION DEFINED amount of the memory set. SETGMT performs an IMPLEMENTATION DEFINED amount of the memory set. SETGET performs the last part of the memory set.

Note

The inclusion of IMPLEMENTATION DEFINED amounts of memory set allows some optimization of the size that can be performed.

The architecture supports two algorithms for the memory set: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

After execution of SETGPT, option A (which results in encoding PSTATE.C = 0):

- If Xn<63> == 1, the set size is saturated to `0x7FFFFFFFFFFFFFF0`.
- Xd holds the original Xd + saturated Xn.
- Xn holds -1* saturated Xn + an IMPLEMENTATION DEFINED number of bytes set.
- PSTATE.{N,Z,V} are set to {0,0,0}.

After execution of SETGPT, option B (which results in encoding PSTATE.C = 1):

- If Xn<63> == 1, the copy size is saturated to `0x7FFFFFFFFFFFFFF0`.
- Xd holds the original Xd + an IMPLEMENTATION DEFINED number of bytes set.
- Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes set.
- PSTATE.{N,Z,V} are set to {0,0,0}.

For SETGMT, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number.
- Xn holds -1* number of bytes remaining to be set in the memory set in total.
- Xd holds the lowest address that the set is made to -Xn.
- At the end of the instruction, the value of Xn is written back with -1* number of bytes remaining to be set in the memory set in total.

For SETGMT, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes remaining to be set in the memory set in total.
- Xd holds the lowest address that the set is made to.
- At the end of the instruction:
  - the value of Xn is written back with the number of bytes remaining to be set in the memory set in total.
  - the value of Xd is written back with the lowest address that has not been set.

For SETGET, option A (encoded by PSTATE.C = 0), the format of the arguments is:

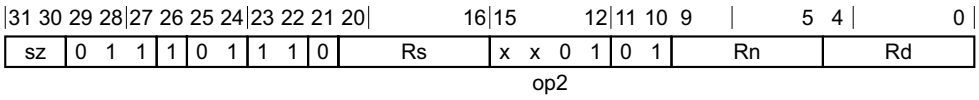
- Xn is treated as a signed 64-bit number.
- Xn holds -1* the number of bytes remaining to be set in the memory set in total.
- Xd holds the lowest address that the set is made to -Xn.
- At the end of the instruction, the value of Xn is written back with 0.

For SETGET, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes remaining to be set in the memory set in total.
- Xd holds the lowest address that the set is made to.
- At the end of the instruction:
  - the value of Xn is written back with 0.
  - the value of Xd is written back with the lowest address that has not been set.

Integer

(FEAT_MOPS)



Epilogue variant

Applies when op2 == 1001.

```
SETGET [<Xd>]!, <Xn>!, <Xs>
```

Main variant

Applies when op2 == 0101.

```
SETGMT [<Xd>]!, <Xn>!, <Xs>
```

Prologue variant

Applies when op2 == 0001.

```
SETGPT [<Xd>]!, <Xn>!, <Xs>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_MOPS) || !IsFeatureImplemented(FEAT_MTE) || sz != '00' then UNDEFINED;

SETParams memset;
memset.d = UInt(Rd);
memset.s = UInt(Rs);
memset.n = UInt(Rn);
bits(2) options = op2<1:0>;
boolean nontemporal = options<1> == '1';

case op2<3:2> of
    when '00' memset.stage = MOPSStage_Prologue;
    when '01' memset.stage = MOPSStage_Main;
    when '10' memset.stage = MOPSStage_Epilogue;
    otherwise UNDEFINED;

CheckMOPSEnabled();
```

```
 if (memset.s == memset.n || memset.s == memset.d || memset.n == memset.d || memset.d == 31 || memset.n
== 31) then
     Constraint c = ConstrainUnpredictable(Unpredictable_MOPSOVERLAP31);
     assert c IN {Constraint_UNDEF, Constraint_NOP};
     case c of
         when Constraint_UNDEF UNDEFINED;
         when Constraint_NOP   EndOfInstruction();
```

Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*, and particularly *Memory Copy and Memory Set SET**.

Assembler symbols

<Xd> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address (an integer multiple of 16) and for option B is updated by the instruction, encoded in the "Rd" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address (an integer multiple of 16) and is updated by the instruction, encoded in the "Rd" field.

<Xn> For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be set (an integer multiple of 16) and is set to zero at the end of the instruction, encoded in the "Rn" field.

For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be set (an integer multiple of 16) and is updated by the instruction, encoded in the "Rn" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be set (an integer multiple of 16) and is updated by the instruction, encoded in the "Rn" field.

<Xs> For the epilogue variant: is the 64-bit name of the general-purpose register that holds the source data, encoded in the "Rs" field.

For the main and prologue variant: is the 64-bit name of the general-purpose register that holds the source data in bits<7:0>, encoded in the "Rs" field.

Operation

```
bits(8) data = X[memset.s, 8];
integer B;

memset.is_setg = TRUE;
memset.nzcv = PSTATE.<N,Z,C,V>;
memset.toaddress = X[memset.d, 64];
if memset.stage == MOPSStage_Prologue then
    memset.setsize = UInt(X[memset.n, 64]);
else
    memset.setsize = SInt(X[memset.n, 64]);
memset.implements_option_a = SETGOptionA();

boolean privileged = if options<0> == '1' then AArch64.IsUnprivAccessPriv() else PSTATE.EL != EL0;

AccessDescriptor accdesc = CreateAccDescSTGMOPS(privileged, nontemporal);

if memset.stage == MOPSStage_Prologue then
    if memset.setsize > 0x7FFFFFFFFFFFFFF0 then
        memset.setsize = 0x7FFFFFFFFFFFFFF0;
```

```
    if ((memset.setsize != 0 && !IsAligned(memset.toaddress, TAG_GRANULE)) ||
!IsAligned(memset.setsize<63:0>, TAG_GRANULE)) then
        AArch64.Abort(memset.toaddress, AlignmentFault(accdesc));

    if memset.implements_option_a then
        memset.nzcv = '0000';
        memset.toaddress = memset.toaddress + memset.setsize;
        memset.setsize = 0 - memset.setsize;
    else
        memset.nzcv = '0010';

memset.stagesetsize = MemSetStageSize(memset);

if memset.stage != MOPSStage_Prologue then
    CheckMemSetParams(memset, options);

    if ((memset.setsize != 0 && !IsAligned(memset.toaddress, TAG_GRANULE)) ||
!IsAligned(memset.setsize<63:0>, TAG_GRANULE)) then
        AArch64.Abort(memset.toaddress, AlignmentFault(accdesc));

integer tagstep;
bits(4) tag;
bits(64) tagaddr;
AddressDescriptor memaddrdesc;
PhysMemRetStatus memstatus;
integer memory_set;
boolean fault = FALSE;

if memset.implements_option_a then
    while memset.stagesetsize < 0 && !fault do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = SETSizeChoice(memset, 16);
        assert B <= -1 * memset.stagesetsize && B<3:0> == '0000';

        (memory_set, memaddrdesc, memstatus) = MemSetBytes(memset.toaddress + memset.setsize, data, B,
accdesc);

        if memory_set != B then
            fault = TRUE;
        else
            tagstep = B DIV 16;
            tag = AArch64.AllocationTagFromAddress(memset.toaddress + memset.setsize);

            while tagstep > 0 do
                tagaddr = memset.toaddress + memset.setsize + (tagstep - 1) * 16;
                AArch64.MemTag[tagaddr, accdesc] = tag;
                tagstep = tagstep - 1;

            memset.setsize = memset.setsize + B;
            memset.stagesetsize = memset.stagesetsize + B;

else
    while memset.stagesetsize > 0 && !fault do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = SETSizeChoice(memset, 16);
        assert B <= memset.stagesetsize && B<3:0> == '0000';

        (memory_set, memaddrdesc, memstatus) = MemSetBytes(memset.toaddress, data, B, accdesc);

        if memory_set != B then
            fault = TRUE;
        else
            tagstep = B DIV 16;
            tag = AArch64.AllocationTagFromAddress(memset.toaddress);
            while tagstep > 0 do
                tagaddr = memset.toaddress + (tagstep - 1) * 16;
```

```
                AArch64.MemTag[tagaddr, accdesc] = tag;
                tagstep = tagstep - 1;

            memset.toaddress = memset.toaddress + B;
            memset.setsize = memset.setsize - B;
            memset.stagesetsize = memset.stagesetsize - B;

UpdateSetRegisters(memset, fault, memory_set);

if fault then
    if IsFault(memaddrdesc) then
        AArch64.Abort(memaddrdesc.vaddress, memaddrdesc.fault);
    else
        boolean iswrite = TRUE;
        HandleExternalAbort(memstatus, iswrite, memaddrdesc, B, accdesc);

if memset.stage == MOPSStage_Prologue then
    PSTATE.<N,Z,C,V> = memset.nzcv;
```

C6.2.313 SETGPTN, SETGMTN, SETGETN

Memory Set with tag setting, unprivileged and non-temporal. These instructions perform a memory set using the value in the bottom byte of the source register and store an Allocation Tag to memory for each Tag Granule written. The Allocation Tag is calculated from the Logical Address Tag in the register which holds the first address that the set is made to. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: SETGPTN, then SETGMTN, and then SETGETN.

SETGPTN performs some preconditioning of the arguments suitable for using the SETGMTN instruction, and performs an IMPLEMENTATION DEFINED amount of the memory set. SETGMTN performs an IMPLEMENTATION DEFINED amount of the memory set. SETGETN performs the last part of the memory set.

Note

The inclusion of IMPLEMENTATION DEFINED amounts of memory set allows some optimization of the size that can be performed.

The architecture supports two algorithms for the memory set: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

After execution of SETGPTN, option A (which results in encoding PSTATE.C = 0):

- If Xn<63> == 1, the set size is saturated to `0x7FFFFFFFFFFFFFF0`.
- Xd holds the original Xd + saturated Xn.
- Xn holds -1* saturated Xn + an IMPLEMENTATION DEFINED number of bytes set.
- PSTATE.{N,Z,V} are set to {0,0,0}.

After execution of SETGPTN, option B (which results in encoding PSTATE.C = 1):

- If Xn<63> == 1, the copy size is saturated to `0x7FFFFFFFFFFFFFF0`.
- Xd holds the original Xd + an IMPLEMENTATION DEFINED number of bytes set.
- Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes set.
- PSTATE.{N,Z,V} are set to {0,0,0}.

For SETGMTN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number.
- Xn holds -1* number of bytes remaining to be set in the memory set in total.
- Xd holds the lowest address that the set is made to -Xn.
- At the end of the instruction, the value of Xn is written back with -1* number of bytes remaining to be set in the memory set in total.

For SETGMTN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes remaining to be set in the memory set in total.
- Xd holds the lowest address that the set is made to.
- At the end of the instruction:
  - the value of Xn is written back with the number of bytes remaining to be set in the memory set in total.
  - the value of Xd is written back with the lowest address that has not been set.

For SETGETN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

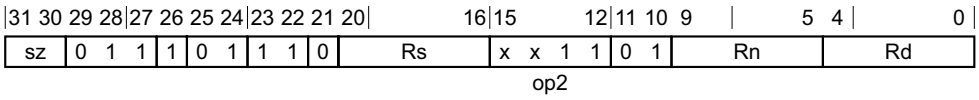
- Xn is treated as a signed 64-bit number.
- Xn holds -1* the number of bytes remaining to be set in the memory set in total.
- Xd holds the lowest address that the set is made to -Xn.
- At the end of the instruction, the value of Xn is written back with 0.

For SETGETN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes remaining to be set in the memory set in total.
- Xd holds the lowest address that the set is made to.
- At the end of the instruction:
  - the value of Xn is written back with 0.
  - the value of Xd is written back with the lowest address that has not been set.

Integer

(FEAT_MOPS)



Epilogue variant

Applies when op2 == 1011.

```
SETGETN [<Xd>]!, <Xn>!, <Xs>
```

Main variant

Applies when op2 == 0111.

```
SETGMTN [<Xd>]!, <Xn>!, <Xs>
```

Prologue variant

Applies when op2 == 0011.

```
SETGPTN [<Xd>]!, <Xn>!, <Xs>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_MOPS) || !IsFeatureImplemented(FEAT_MTE) || sz != '00' then UNDEFINED;

SETParams memset;
memset.d = UInt(Rd);
memset.s = UInt(Rs);
memset.n = UInt(Rn);
bits(2) options = op2<1:0>;
boolean nontemporal = options<1> == '1';

case op2<3:2> of
    when '00' memset.stage = MOPSStage_Prologue;
    when '01' memset.stage = MOPSStage_Main;
    when '10' memset.stage = MOPSStage_Epilogue;
    otherwise UNDEFINED;

CheckMOPSEnabled();
```

```
 if (memset.s == memset.n || memset.s == memset.d || memset.n == memset.d || memset.d == 31 || memset.n
== 31) then
     Constraint c = ConstrainUnpredictable(Unpredictable_MOPSOVERLAP31);
     assert c IN {Constraint_UNDEF, Constraint_NOP};
     case c of
         when Constraint_UNDEF UNDEFINED;
         when Constraint_NOP   EndOfInstruction();
```

Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*, and particularly *Memory Copy and Memory Set SET**.

Assembler symbols

<Xd> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address (an integer multiple of 16) and for option B is updated by the instruction, encoded in the "Rd" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address (an integer multiple of 16) and is updated by the instruction, encoded in the "Rd" field.

<Xn> For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be set (an integer multiple of 16) and is set to zero at the end of the instruction, encoded in the "Rn" field.

For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be set (an integer multiple of 16) and is updated by the instruction, encoded in the "Rn" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be set (an integer multiple of 16) and is updated by the instruction, encoded in the "Rn" field.

<Xs> For the epilogue variant: is the 64-bit name of the general-purpose register that holds the source data, encoded in the "Rs" field.

For the main and prologue variant: is the 64-bit name of the general-purpose register that holds the source data in bits<7:0>, encoded in the "Rs" field.

Operation

```
bits(8) data = X[memset.s, 8];
integer B;

memset.is_setg = TRUE;
memset.nzcv = PSTATE.<N,Z,C,V>;
memset.toaddress = X[memset.d, 64];
if memset.stage == MOPSStage_Prologue then
    memset.setsize = UInt(X[memset.n, 64]);
else
    memset.setsize = SInt(X[memset.n, 64]);
memset.implements_option_a = SETGOptionA();

boolean privileged = if options<0> == '1' then AArch64.IsUnprivAccessPriv() else PSTATE.EL != EL0;

AccessDescriptor accdesc = CreateAccDescSTGMOPS(privileged, nontemporal);

if memset.stage == MOPSStage_Prologue then
    if memset.setsize > 0x7FFFFFFFFFFFFFF0 then
        memset.setsize = 0x7FFFFFFFFFFFFFF0;
```

```
    if ((memset.setsize != 0 && !IsAligned(memset.toaddress, TAG_GRANULE)) ||
!IsAligned(memset.setsize<63:0>, TAG_GRANULE)) then
        AArch64.Abort(memset.toaddress, AlignmentFault(accdesc));

    if memset.implements_option_a then
        memset.nzcv = '0000';
        memset.toaddress = memset.toaddress + memset.setsize;
        memset.setsize = 0 - memset.setsize;
    else
        memset.nzcv = '0010';

memset.stagesetsize = MemSetStageSize(memset);

if memset.stage != MOPSStage_Prologue then
    CheckMemSetParams(memset, options);

    if ((memset.setsize != 0 && !IsAligned(memset.toaddress, TAG_GRANULE)) ||
!IsAligned(memset.setsize<63:0>, TAG_GRANULE)) then
        AArch64.Abort(memset.toaddress, AlignmentFault(accdesc));

integer tagstep;
bits(4) tag;
bits(64) tagaddr;
AddressDescriptor memaddrdesc;
PhysMemRetStatus memstatus;
integer memory_set;
boolean fault = FALSE;

if memset.implements_option_a then
    while memset.stagesetsize < 0 && !fault do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = SETSizeChoice(memset, 16);
        assert B <= -1 * memset.stagesetsize && B<3:0> == '0000';

        (memory_set, memaddrdesc, memstatus) = MemSetBytes(memset.toaddress + memset.setsize, data, B,
accdesc);

        if memory_set != B then
            fault = TRUE;
        else
            tagstep = B DIV 16;
            tag = AArch64.AllocationTagFromAddress(memset.toaddress + memset.setsize);

            while tagstep > 0 do
                tagaddr = memset.toaddress + memset.setsize + (tagstep - 1) * 16;
                AArch64.MemTag[tagaddr, accdesc] = tag;
                tagstep = tagstep - 1;

            memset.setsize = memset.setsize + B;
            memset.stagesetsize = memset.stagesetsize + B;

else
    while memset.stagesetsize > 0 && !fault do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = SETSizeChoice(memset, 16);
        assert B <= memset.stagesetsize && B<3:0> == '0000';

        (memory_set, memaddrdesc, memstatus) = MemSetBytes(memset.toaddress, data, B, accdesc);

        if memory_set != B then
            fault = TRUE;
        else
            tagstep = B DIV 16;
            tag = AArch64.AllocationTagFromAddress(memset.toaddress);
            while tagstep > 0 do
                tagaddr = memset.toaddress + (tagstep - 1) * 16;
```

```
                AArch64.MemTag[tagaddr, accdesc] = tag;
                tagstep = tagstep - 1;

            memset.toaddress = memset.toaddress + B;
            memset.setsize = memset.setsize - B;
            memset.stagesetsize = memset.stagesetsize - B;

UpdateSetRegisters(memset, fault, memory_set);

if fault then
    if IsFault(memaddrdesc) then
        AArch64.Abort(memaddrdesc.vaddress, memaddrdesc.fault);
    else
        boolean iswrite = TRUE;
        HandleExternalAbort(memstatus, iswrite, memaddrdesc, B, accdesc);

if memset.stage == MOPSStage_Prologue then
    PSTATE.<N,Z,C,V> = memset.nzcv;
```

C6.2.314 SETP, SETM, SETE

Memory Set. These instructions perform a memory set using the value in the bottom byte of the source register. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: SETP, then SETM, and then SETE.

SETP performs some preconditioning of the arguments suitable for using the SETM instruction, and performs an IMPLEMENTATION DEFINED amount of the memory set. SETM performs an IMPLEMENTATION DEFINED amount of the memory set. SETE performs the last part of the memory set.

Note

The inclusion of IMPLEMENTATION DEFINED amounts of memory set allows some optimization of the size that can be performed.

The architecture supports two algorithms for the memory set: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

After execution of SETP, option A (which results in encoding PSTATE.C = 0):

- If Xn<63> == 1, the set size is saturated to `0x7FFFFFFFFFFFFFFF`.
- Xd holds the original Xd + saturated Xn.
- Xn holds -1* saturated Xn + an IMPLEMENTATION DEFINED number of bytes set.
- PSTATE.{N,Z,V} are set to {0,0,0}.

After execution of SETP, option B (which results in encoding PSTATE.C = 1):

- If Xn<63> == 1, the copy size is saturated to `0x7FFFFFFFFFFFFFFF`.
- Xd holds the original Xd + an IMPLEMENTATION DEFINED number of bytes set.
- Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes set.
- PSTATE.{N,Z,V} are set to {0,0,0}.

For SETM, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number.
- Xn holds -1* number of bytes remaining to be set in the memory set in total.
- Xd holds the lowest address that the set is made to -Xn.
- At the end of the instruction, the value of Xn is written back with -1* the number of bytes remaining to be set in the memory set in total.

For SETM, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes remaining to be set in the memory set in total.
- Xd holds the lowest address that the set is made to.
- At the end of the instruction:
  - the value of Xn is written back with the number of bytes remaining to be set in the memory set in total.
  - the value of Xd is written back with the lowest address that has not been set.

For SETE, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number.

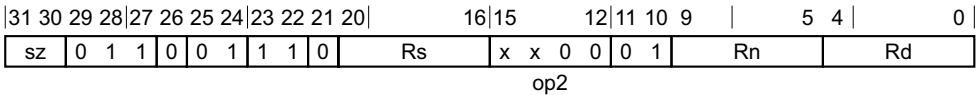
- Xn holds -1* the number of bytes remaining to be set in the memory set in total.
- Xd holds the lowest address that the set is made to -Xn.
- At the end of the instruction, the value of Xn is written back with 0.

For SETE, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes remaining to be set in the memory set in total.
- Xd holds the lowest address that the set is made to.
- At the end of the instruction:
  - the value of Xn is written back with 0.
  - the value of Xd is written back with the lowest address that has not been set.

Integer

(FEAT_MOPS)



Epilogue variant

Applies when op2 == 1000.

```
SETE [<Xd>]!, <Xn>!, <Xs>
```

Main variant

Applies when op2 == 0100.

```
SETM [<Xd>]!, <Xn>!, <Xs>
```

Prologue variant

Applies when op2 == 0000.

```
SETP [<Xd>]!, <Xn>!, <Xs>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_MOPS) || sz != '00' then UNDEFINED;

SETParams memset;
memset.d = UInt(Rd);
memset.s = UInt(Rs);
memset.n = UInt(Rn);
bits(2) options = op2<1:0>;
boolean nontemporal = options<1> == '1';

case op2<3:2> of
    when '00' memset.stage = MOPSStage_Prologue;
    when '01' memset.stage = MOPSStage_Main;
    when '10' memset.stage = MOPSStage_Epilogue;
    otherwise UNDEFINED;

CheckMOPSEnabled();

if (memset.s == memset.n || memset.s == memset.d || memset.n == memset.d || memset.d == 31 || memset.n
== 31) then
    Constraint c = ConstrainUnpredictable(Unpredictable_MOPSOVERLAP31);
```

```
    assert c IN {Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNDEF UNDEFINED;
        when Constraint_NOP   EndOfInstruction();
```

Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*, and particularly *Memory Copy and Memory Set SET**.

Assembler symbols

<Xd> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address and for option B is updated by the instruction, encoded in the "Rd" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.

<Xn> For the epilogue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be set and is set to zero at the end of the instruction, encoded in the "Rn" field.

For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be set and is updated by the instruction, encoded in the "Rn" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be set and is updated by the instruction, encoded in the "Rn" field.

<Xs> Is the 64-bit name of the general-purpose register that holds the source data, encoded in the "Rs" field.

Operation

```
bits(8) data = X[memset.s, 8];
integer B;

memset.is_setg = FALSE;
memset.nzcv = PSTATE.<N,Z,C,V>;
memset.toaddress = X[memset.d, 64];
if memset.stage == MOPSStage_Prologue then
    memset.setsize = UInt(X[memset.n, 64]);
else
    memset.setsize = SInt(X[memset.n, 64]);
memset.implements_option_a = SETOptionA();

boolean privileged = if options<0> == '1' then AArch64.IsUnprivAccessPriv() else PSTATE.EL != EL0;

AccessDescriptor accdesc = CreateAccDescMOPS(MemOp_STORE, privileged, nontemporal);

if memset.stage == MOPSStage_Prologue then
    if memset.setsize > 0x7FFFFFFFFFFFFFFF then
        memset.setsize = 0x7FFFFFFFFFFFFFFF;

    if memset.implements_option_a then
        memset.nzcv = '0000';
        memset.toaddress = memset.toaddress + memset.setsize;
        memset.setsize = 0 - memset.setsize;
    else
        memset.nzcv = '0010';

memset.stagesetsize = MemSetStageSize(memset);

if memset.stage != MOPSStage_Prologue then
    CheckMemSetParams(memset, options);
```

```
AddressDescriptor memaddrdesc;
PhysMemRetStatus memstatus;
integer memory_set;
boolean fault = FALSE;

if memset.implements_option_a then
    while memset.stagesetsize < 0 && !fault do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = SETSizeChoice(memset, 1);
        assert B <= -1 * memset.stagesetsize;

        (memory_set, memaddrdesc, memstatus) = MemSetBytes(memset.toaddress + memset.setsize, data, B,
accdesc);

        if memory_set != B then
            fault = TRUE;
        else
            memset.setsize = memset.setsize + B;
            memset.stagesetsize = memset.stagesetsize + B;

else
    while memset.stagesetsize > 0 && !fault do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = SETSizeChoice(memset, 1);
        assert B <= memset.stagesetsize;

        (memory_set, memaddrdesc, memstatus) = MemSetBytes(memset.toaddress, data, B, accdesc);

        if memory_set != B then
            fault = TRUE;
        else
            memset.toaddress = memset.toaddress + B;
            memset.setsize = memset.setsize - B;
            memset.stagesetsize = memset.stagesetsize - B;

UpdateSetRegisters(memset, fault, memory_set);

if fault then
    if IsFault(memaddrdesc) then
        AArch64.Abort(memaddrdesc.vaddress, memaddrdesc.fault);
    else
        boolean iswrite = TRUE;
        HandleExternalAbort(memstatus, iswrite, memaddrdesc, B, accdesc);

if memset.stage == MOPSStage_Prologue then
    PSTATE.<N,Z,C,V> = memset.nzcv;
```

C6.2.315 SETPN, SETMN, SETEN

Memory Set, non-temporal. These instructions perform a memory set using the value in the bottom byte of the source register. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: SETPN, then SETMN, and then SETEN.

SETPN performs some preconditioning of the arguments suitable for using the SETMN instruction, and performs an IMPLEMENTATION DEFINED amount of the memory set. SETMN performs an IMPLEMENTATION DEFINED amount of the memory set. SETEN performs the last part of the memory set.

Note

The inclusion of IMPLEMENTATION DEFINED amounts of memory set allows some optimization of the size that can be performed.

The architecture supports two algorithms for the memory set: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

After execution of SETPN, option A (which results in encoding PSTATE.C = 0):

- If Xn<63> == 1, the set size is saturated to `0x7FFFFFFFFFFFFFFF`.
- Xd holds the original Xd + saturated Xn.
- Xn holds -1* saturated Xn + an IMPLEMENTATION DEFINED number of bytes set.
- PSTATE.{N,Z,V} are set to {0,0,0}.

After execution of SETPN, option B (which results in encoding PSTATE.C = 1):

- If Xn<63> == 1, the copy size is saturated to `0x7FFFFFFFFFFFFFFF`.
- Xd holds the original Xd + an IMPLEMENTATION DEFINED number of bytes set.
- Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes set.
- PSTATE.{N,Z,V} are set to {0,0,0}.

For SETMN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number.
- Xn holds -1* number of bytes remaining to be set in the memory set in total.
- Xd holds the lowest address that the set is made to -Xn.
- At the end of the instruction, the value of Xn is written back with -1* the number of bytes remaining to be set in the memory set in total.

For SETMN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes remaining to be set in the memory set in total.
- Xd holds the lowest address that the set is made to.
- At the end of the instruction:
  - the value of Xn is written back with the number of bytes remaining to be set in the memory set in total.
  - the value of Xd is written back with the lowest address that has not been set.

For SETEN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number.

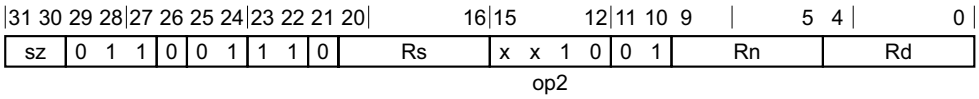
- Xn holds -1* the number of bytes remaining to be set in the memory set in total.
- Xd holds the lowest address that the set is made to -Xn.
- At the end of the instruction, the value of Xn is written back with 0.

For SETEN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes remaining to be set in the memory set in total.
- Xd holds the lowest address that the set is made to.
- At the end of the instruction:
  - the value of Xn is written back with 0.
  - the value of Xd is written back with the lowest address that has not been set.

Integer

(FEAT_MOPS)



Epilogue variant

Applies when op2 == 1010.

```
SETEN [<Xd>]!, <Xn>!, <Xs>
```

Main variant

Applies when op2 == 0110.

```
SETMN [<Xd>]!, <Xn>!, <Xs>
```

Prologue variant

Applies when op2 == 0010.

```
SETPN [<Xd>]!, <Xn>!, <Xs>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_MOPS) || sz != '00' then UNDEFINED;

SETParams memset;
memset.d = UInt(Rd);
memset.s = UInt(Rs);
memset.n = UInt(Rn);
bits(2) options = op2<1:0>;
boolean nontemporal = options<1> == '1';

case op2<3:2> of
    when '00' memset.stage = MOPSStage_Prologue;
    when '01' memset.stage = MOPSStage_Main;
    when '10' memset.stage = MOPSStage_Epilogue;
    otherwise UNDEFINED;

CheckMOPSEnabled();

if (memset.s == memset.n || memset.s == memset.d || memset.n == memset.d || memset.d == 31 || memset.n
== 31) then
    Constraint c = ConstrainUnpredictable(Unpredictable_MOPSOVERLAP31);
```

```
    assert c IN {Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNDEF UNDEFINED;
        when Constraint_NOP   EndOfInstruction();
```

Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*, and particularly *Memory Copy and Memory Set SET**.

Assembler symbols

<Xd> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address and for option B is updated by the instruction, encoded in the "Rd" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.

<Xn> For the epilogue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be set and is set to zero at the end of the instruction, encoded in the "Rn" field.

For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be set and is updated by the instruction, encoded in the "Rn" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be set and is updated by the instruction, encoded in the "Rn" field.

<Xs> Is the 64-bit name of the general-purpose register that holds the source data, encoded in the "Rs" field.

Operation

```
bits(8) data = X[memset.s, 8];
integer B;

memset.is_setg = FALSE;
memset.nzcv = PSTATE.<N,Z,C,V>;
memset.toaddress = X[memset.d, 64];
if memset.stage == MOPSStage_Prologue then
    memset.setsize = UInt(X[memset.n, 64]);
else
    memset.setsize = SInt(X[memset.n, 64]);
memset.implements_option_a = SETOptionA();

boolean privileged = if options<0> == '1' then AArch64.IsUnprivAccessPriv() else PSTATE.EL != EL0;

AccessDescriptor accdesc = CreateAccDescMOPS(MemOp_STORE, privileged, nontemporal);

if memset.stage == MOPSStage_Prologue then
    if memset.setsize > 0x7FFFFFFFFFFFFFFF then
        memset.setsize = 0x7FFFFFFFFFFFFFFF;

    if memset.implements_option_a then
        memset.nzcv = '0000';
        memset.toaddress = memset.toaddress + memset.setsize;
        memset.setsize = 0 - memset.setsize;
    else
        memset.nzcv = '0010';

memset.stagesetsize = MemSetStageSize(memset);

if memset.stage != MOPSStage_Prologue then
    CheckMemSetParams(memset, options);
```

```
AddressDescriptor memaddrdesc;
PhysMemRetStatus memstatus;
integer memory_set;
boolean fault = FALSE;

if memset.implements_option_a then
    while memset.stagesetsize < 0 && !fault do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = SETSizeChoice(memset, 1);
        assert B <= -1 * memset.stagesetsize;

        (memory_set, memaddrdesc, memstatus) = MemSetBytes(memset.toaddress + memset.setsize, data, B,
accdesc);

        if memory_set != B then
            fault = TRUE;
        else
            memset.setsize = memset.setsize + B;
            memset.stagesetsize = memset.stagesetsize + B;

else
    while memset.stagesetsize > 0 && !fault do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = SETSizeChoice(memset, 1);
        assert B <= memset.stagesetsize;

        (memory_set, memaddrdesc, memstatus) = MemSetBytes(memset.toaddress, data, B, accdesc);

        if memory_set != B then
            fault = TRUE;
        else
            memset.toaddress = memset.toaddress + B;
            memset.setsize = memset.setsize - B;
            memset.stagesetsize = memset.stagesetsize - B;

UpdateSetRegisters(memset, fault, memory_set);

if fault then
    if IsFault(memaddrdesc) then
        AArch64.Abort(memaddrdesc.vaddress, memaddrdesc.fault);
    else
        boolean iswrite = TRUE;
        HandleExternalAbort(memstatus, iswrite, memaddrdesc, B, accdesc);

if memset.stage == MOPSStage_Prologue then
    PSTATE.<N,Z,C,V> = memset.nzcv;
```

C6.2.316 SETPT, SETMT, SETET

Memory Set, unprivileged. These instructions perform a memory set using the value in the bottom byte of the source register. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: SETPT, then SETMT, and then SETET.

SETPT performs some preconditioning of the arguments suitable for using the SETMT instruction, and performs an IMPLEMENTATION DEFINED amount of the memory set. SETMT performs an IMPLEMENTATION DEFINED amount of the memory set. SETET performs the last part of the memory set.

Note

The inclusion of IMPLEMENTATION DEFINED amounts of memory set allows some optimization of the size that can be performed.

The architecture supports two algorithms for the memory set: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

After execution of SETPT, option A (which results in encoding PSTATE.C = 0):

- If Xn<63> == 1, the set size is saturated to `0x7FFFFFFFFFFFFFFF`.
- Xd holds the original Xd + saturated Xn.
- Xn holds -1* saturated Xn + an IMPLEMENTATION DEFINED number of bytes set.
- PSTATE.{N,Z,V} are set to {0,0,0}.

After execution of SETPT, option B (which results in encoding PSTATE.C = 1):

- If Xn<63> == 1, the copy size is saturated to `0x7FFFFFFFFFFFFFFF`.
- Xd holds the original Xd + an IMPLEMENTATION DEFINED number of bytes set.
- Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes set.
- PSTATE.{N,Z,V} are set to {0,0,0}.

For SETMT, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number.
- Xn holds -1* number of bytes remaining to be set in the memory set in total.
- Xd holds the lowest address that the set is made to -Xn.
- At the end of the instruction, the value of Xn is written back with -1* the number of bytes remaining to be set in the memory set in total.

For SETMT, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes remaining to be set in the memory set in total.
- Xd holds the lowest address that the set is made to.
- At the end of the instruction:
  - the value of Xn is written back with the number of bytes remaining to be set in the memory set in total.
  - the value of Xd is written back with the lowest address that has not been set.

For SETET, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number.

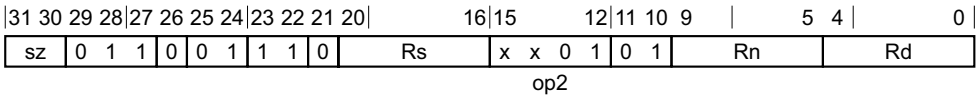
- Xn holds -1* the number of bytes remaining to be set in the memory set in total.
- Xd holds the lowest address that the set is made to -Xn.
- At the end of the instruction, the value of Xn is written back with 0.

For SETET, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes remaining to be set in the memory set in total.
- Xd holds the lowest address that the set is made to.
- At the end of the instruction:
  - the value of Xn is written back with 0.
  - the value of Xd is written back with the lowest address that has not been set.

Integer

(FEAT_MOPS)



Epilogue variant

Applies when op2 == 1001.

```
SETET [<Xd>]!, <Xn>!, <Xs>
```

Main variant

Applies when op2 == 0101.

```
SETMT [<Xd>]!, <Xn>!, <Xs>
```

Prologue variant

Applies when op2 == 0001.

```
SETPT [<Xd>]!, <Xn>!, <Xs>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_MOPS) || sz != '00' then UNDEFINED;

SETParams memset;
memset.d = UInt(Rd);
memset.s = UInt(Rs);
memset.n = UInt(Rn);
bits(2) options = op2<1:0>;
boolean nontemporal = options<1> == '1';

case op2<3:2> of
    when '00' memset.stage = MOPSStage_Prologue;
    when '01' memset.stage = MOPSStage_Main;
    when '10' memset.stage = MOPSStage_Epilogue;
    otherwise UNDEFINED;

CheckMOPSEnabled();

if (memset.s == memset.n || memset.s == memset.d || memset.n == memset.d || memset.d == 31 || memset.n
== 31) then
    Constraint c = ConstrainUnpredictable(Unpredictable_MOPSOVERLAP31);
```

```
    assert c IN {Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNDEF UNDEFINED;
        when Constraint_NOP   EndOfInstruction();
```

Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*, and particularly *Memory Copy and Memory Set SET**.

Assembler symbols

<Xd> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address and for option B is updated by the instruction, encoded in the "Rd" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.

<Xn> For the epilogue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be set and is set to zero at the end of the instruction, encoded in the "Rn" field.

For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be set and is updated by the instruction, encoded in the "Rn" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be set and is updated by the instruction, encoded in the "Rn" field.

<Xs> Is the 64-bit name of the general-purpose register that holds the source data, encoded in the "Rs" field.

Operation

```
bits(8) data = X[memset.s, 8];
integer B;

memset.is_setg = FALSE;
memset.nzcv = PSTATE.<N,Z,C,V>;
memset.toaddress = X[memset.d, 64];
if memset.stage == MOPSStage_Prologue then
    memset.setsize = UInt(X[memset.n, 64]);
else
    memset.setsize = SInt(X[memset.n, 64]);
memset.implements_option_a = SETOptionA();

boolean privileged = if options<0> == '1' then AArch64.IsUnprivAccessPriv() else PSTATE.EL != EL0;

AccessDescriptor accdesc = CreateAccDescMOPS(MemOp_STORE, privileged, nontemporal);

if memset.stage == MOPSStage_Prologue then
    if memset.setsize > 0x7FFFFFFFFFFFFFFF then
        memset.setsize = 0x7FFFFFFFFFFFFFFF;

    if memset.implements_option_a then
        memset.nzcv = '0000';
        memset.toaddress = memset.toaddress + memset.setsize;
        memset.setsize = 0 - memset.setsize;
    else
        memset.nzcv = '0010';

memset.stagesetsize = MemSetStageSize(memset);

if memset.stage != MOPSStage_Prologue then
    CheckMemSetParams(memset, options);
```

```
AddressDescriptor memaddrdesc;
PhysMemRetStatus memstatus;
integer memory_set;
boolean fault = FALSE;

if memset.implements_option_a then
    while memset.stagesetsize < 0 && !fault do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = SETSizeChoice(memset, 1);
        assert B <= -1 * memset.stagesetsize;

        (memory_set, memaddrdesc, memstatus) = MemSetBytes(memset.toaddress + memset.setsize, data, B,
accdesc);

        if memory_set != B then
            fault = TRUE;
        else
            memset.setsize = memset.setsize + B;
            memset.stagesetsize = memset.stagesetsize + B;

else
    while memset.stagesetsize > 0 && !fault do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = SETSizeChoice(memset, 1);
        assert B <= memset.stagesetsize;

        (memory_set, memaddrdesc, memstatus) = MemSetBytes(memset.toaddress, data, B, accdesc);

        if memory_set != B then
            fault = TRUE;
        else
            memset.toaddress = memset.toaddress + B;
            memset.setsize = memset.setsize - B;
            memset.stagesetsize = memset.stagesetsize - B;

UpdateSetRegisters(memset, fault, memory_set);

if fault then
    if IsFault(memaddrdesc) then
        AArch64.Abort(memaddrdesc.vaddress, memaddrdesc.fault);
    else
        boolean iswrite = TRUE;
        HandleExternalAbort(memstatus, iswrite, memaddrdesc, B, accdesc);

if memset.stage == MOPSStage_Prologue then
    PSTATE.<N,Z,C,V> = memset.nzcv;
```

C6.2.317 SETPTN, SETMTN, SETETN

Memory Set, unprivileged and non-temporal. These instructions perform a memory set using the value in the bottom byte of the source register. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: SETPTN, then SETMTN, and then SETETN.

SETPTN performs some preconditioning of the arguments suitable for using the SETMTN instruction, and performs an IMPLEMENTATION DEFINED amount of the memory set. SETMTN performs an IMPLEMENTATION DEFINED amount of the memory set. SETETN performs the last part of the memory set.

Note

The inclusion of IMPLEMENTATION DEFINED amounts of memory set allows some optimization of the size that can be performed.

The architecture supports two algorithms for the memory set: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

After execution of SETPTN, option A (which results in encoding PSTATE.C = 0):

- If Xn<63> == 1, the set size is saturated to `0x7FFFFFFFFFFFFFFF`.
- Xd holds the original Xd + saturated Xn.
- Xn holds -1* saturated Xn + an IMPLEMENTATION DEFINED number of bytes set.
- PSTATE.{N,Z,V} are set to {0,0,0}.

After execution of SETPTN, option B (which results in encoding PSTATE.C = 1):

- If Xn<63> == 1, the copy size is saturated to `0x7FFFFFFFFFFFFFFF`.
- Xd holds the original Xd + an IMPLEMENTATION DEFINED number of bytes set.
- Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes set.
- PSTATE.{N,Z,V} are set to {0,0,0}.

For SETMTN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number.
- Xn holds -1* number of bytes remaining to be set in the memory set in total.
- Xd holds the lowest address that the set is made to -Xn.
- At the end of the instruction, the value of Xn is written back with -1* the number of bytes remaining to be set in the memory set in total.

For SETMTN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes remaining to be set in the memory set in total.
- Xd holds the lowest address that the set is made to.
- At the end of the instruction:
  - the value of Xn is written back with the number of bytes remaining to be set in the memory set in total.
  - the value of Xd is written back with the lowest address that has not been set.

For SETETN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number.

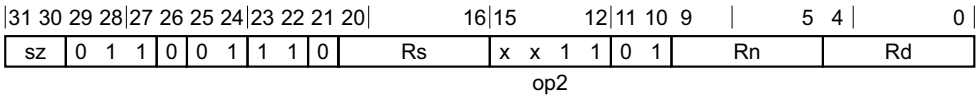
- Xn holds -1* the number of bytes remaining to be set in the memory set in total.
- Xd holds the lowest address that the set is made to -Xn.
- At the end of the instruction, the value of Xn is written back with 0.

For SETETN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes remaining to be set in the memory set in total.
- Xd holds the lowest address that the set is made to.
- At the end of the instruction:
  - the value of Xn is written back with 0.
  - the value of Xd is written back with the lowest address that has not been set.

Integer

(FEAT_MOPS)



Epilogue variant

Applies when op2 == 1011.

```
SETETN [<Xd>]!, <Xn>!, <Xs>
```

Main variant

Applies when op2 == 0111.

```
SETMTN [<Xd>]!, <Xn>!, <Xs>
```

Prologue variant

Applies when op2 == 0011.

```
SETPTN [<Xd>]!, <Xn>!, <Xs>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_MOPS) || sz != '00' then UNDEFINED;

SETParams memset;
memset.d = UInt(Rd);
memset.s = UInt(Rs);
memset.n = UInt(Rn);
bits(2) options = op2<1:0>;
boolean nontemporal = options<1> == '1';

case op2<3:2> of
    when '00' memset.stage = MOPSStage_Prologue;
    when '01' memset.stage = MOPSStage_Main;
    when '10' memset.stage = MOPSStage_Epilogue;
    otherwise UNDEFINED;

CheckMOPSEnabled();

if (memset.s == memset.n || memset.s == memset.d || memset.n == memset.d || memset.d == 31 || memset.n
== 31) then
    Constraint c = ConstrainUnpredictable(Unpredictable_MOPSOVERLAP31);
```

```
    assert c IN {Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNDEF UNDEFINED;
        when Constraint_NOP   EndOfInstruction();
```

Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*, and particularly *Memory Copy and Memory Set SET**.

Assembler symbols

<Xd> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address and for option B is updated by the instruction, encoded in the "Rd" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.

<Xn> For the epilogue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be set and is set to zero at the end of the instruction, encoded in the "Rn" field.

For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be set and is updated by the instruction, encoded in the "Rn" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be set and is updated by the instruction, encoded in the "Rn" field.

<Xs> Is the 64-bit name of the general-purpose register that holds the source data, encoded in the "Rs" field.

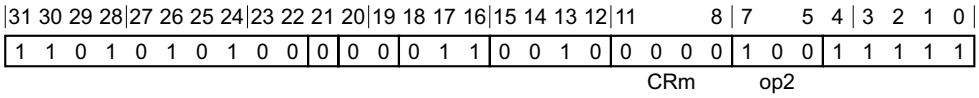
Operation

```
bits(8) data = X[memset.s, 8];
integer B;

memset.is_setg = FALSE;
memset.nzcv = PSTATE.<N,Z,C,V>;
memset.toaddress = X[memset.d, 64];
if memset.stage == MOPSStage_Prologue then
    memset.setsize = UInt(X[memset.n, 64]);
else
    memset.setsize = SInt(X[memset.n, 64]);
memset.implements_option_a = SETOptionA();

boolean privileged = if options<0> == '1' then AArch64.IsUnprivAccessPriv() else PSTATE.EL != EL0;

AccessDescriptor accdesc = CreateAccDescMOPS(MemOp_STORE, privileged, nontemporal);

if memset.stage == MOPSStage_Prologue then
    if memset.setsize > 0x7FFFFFFFFFFFFFFF then
        memset.setsize = 0x7FFFFFFFFFFFFFFF;

    if memset.implements_option_a then
        memset.nzcv = '0000';
        memset.toaddress = memset.toaddress + memset.setsize;
        memset.setsize = 0 - memset.setsize;
    else
        memset.nzcv = '0010';

memset.stagesetsize = MemSetStageSize(memset);

if memset.stage != MOPSStage_Prologue then
    CheckMemSetParams(memset, options);
```

```
AddressDescriptor memaddrdesc;
PhysMemRetStatus memstatus;
integer memory_set;
boolean fault = FALSE;

if memset.implements_option_a then
    while memset.stagesetsize < 0 && !fault do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = SETSizeChoice(memset, 1);
        assert B <= -1 * memset.stagesetsize;

        (memory_set, memaddrdesc, memstatus) = MemSetBytes(memset.toaddress + memset.setsize, data, B,
accdesc);

        if memory_set != B then
            fault = TRUE;
        else
            memset.setsize = memset.setsize + B;
            memset.stagesetsize = memset.stagesetsize + B;

else
    while memset.stagesetsize > 0 && !fault do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = SETSizeChoice(memset, 1);
        assert B <= memset.stagesetsize;

        (memory_set, memaddrdesc, memstatus) = MemSetBytes(memset.toaddress, data, B, accdesc);

        if memory_set != B then
            fault = TRUE;
        else
            memset.toaddress = memset.toaddress + B;
            memset.setsize = memset.setsize - B;
            memset.stagesetsize = memset.stagesetsize - B;

UpdateSetRegisters(memset, fault, memory_set);

if fault then
    if IsFault(memaddrdesc) then
        AArch64.Abort(memaddrdesc.vaddress, memaddrdesc.fault);
    else
        boolean iswrite = TRUE;
        HandleExternalAbort(memstatus, iswrite, memaddrdesc, B, accdesc);

if memset.stage == MOPSStage_Prologue then
    PSTATE.<N,Z,C,V> = memset.nzcv;
```

C6.2.318 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*.



Encoding

SEV

Decode for this encoding

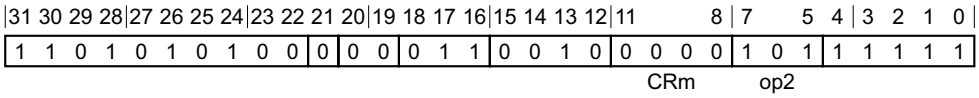
```
// Empty.
```

Operation

```
SendEvent();
```

C6.2.319 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.



Encoding

`SEVL`

Decode for this encoding

```
// Empty.
```

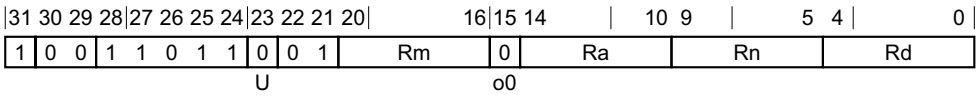
Operation

```
SendEventLocal();
```

C6.2.320 SMADDL

Signed Multiply-Add Long multiplies two 32-bit register values, adds a 64-bit register value, and writes the result to the 64-bit destination register.

This instruction is used by the alias SMULL. See *Alias conditions* for details of when each alias is preferred.



Encoding

```
SMADDL <Xd>, <Wn>, <Wm>, <Xa>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer a = UInt(Ra);
```

Alias conditions

Alias	is preferred when
SMULL	`Ra == '11111'`

Assembler symbols

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Wn> Is the 32-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.

<Wm> Is the 32-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.

<Xa> Is the 64-bit name of the third general-purpose source register holding the addend, encoded in the "Ra" field.

Operation

```
bits(32) operand1 = X[n, 32];
bits(32) operand2 = X[m, 32];
bits(64) operand3 = X[a, 64];

integer result;

result = Int(operand3, FALSE) + (Int(operand1, FALSE) * Int(operand2, FALSE));

X[d, 64] = result<63:0>;
```

Operational information

If PSTATE.DIT is 1:

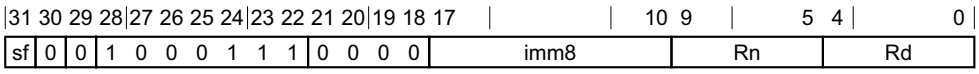
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

C6.2.321 SMAX (immediate)

Signed Maximum (immediate) determines the signed maximum of the source register value and immediate, and writes the result to the destination register.

Integer

(FEAT_CSSC)



32-bit variant

Applies when `sf == 0`.

```
SMAX <Wd>, <Wn>, #<simm>
```

64-bit variant

Applies when `sf == 1`.

```
SMAX <Xd>, <Xn>, #<simm>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_CSSC) then UNDEFINED;
constant integer datasize = 32 << UInt(sf);
integer n = UInt(Rn);
integer d = UInt(Rd);
integer imm = SInt(imm8);
```

Assembler symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
<simm>	Is a signed immediate, in the range -128 to 127, encoded in the "imm8" field.

Operation

```
bits(datasize) operand1 = X[n, datasize];
integer result = Max(SInt(operand1), imm);
X[d, datasize] = result<datasize-1:0>;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

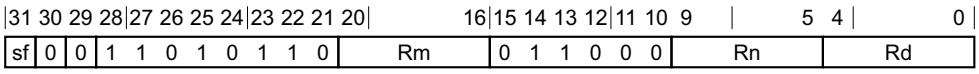
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

C6.2.322 SMAX (register)

Signed Maximum (register) determines the signed maximum of the two source register values and writes the result to the destination register.

Integer

(FEAT_CSSC)



32-bit variant

Applies when `sf == 0`.

```
SMAX <Wd>, <Wn>, <Wm>
```

64-bit variant

Applies when `sf == 1`.

```
SMAX <Xd>, <Xn>, <Xm>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_CSSC) then UNDEFINED;
constant integer datasize = 32 << UInt(sf);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer d = UInt(Rd);
```

Assembler symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Wm>	Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.

Operation

```
bits(datasize) operand1 = X[n, datasize];
bits(datasize) operand2 = X[m, datasize];
integer result = Max(SInt(operand1), SInt(operand2));
X[d, datasize] = result<datasize-1:0>;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.

  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

C6.2.323 SMC

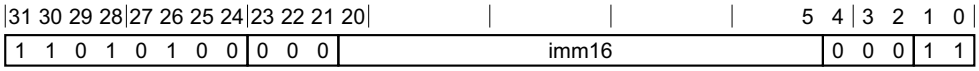
Secure Monitor Call causes an exception to EL3.

SMC is available only for software executing at EL1 or higher. It is UNDEFINED in EL0.

If the values of HCR_EL2.TSC and SCR_EL3.SMD are both 0, execution of an SMC instruction at EL1 or higher generates a Secure Monitor Call exception, recording it in ESR_ELx, using the EC value 0x17, that is taken to EL3.

If the value of HCR_EL2.TSC is 1 and EL2 is enabled in the current Security state, execution of an SMC instruction at EL1 generates an exception that is taken to EL2, regardless of the value of SCR_EL3.SMD.

If the value of HCR_EL2.TSC is 0 and the value of SCR_EL3.SMD is 1, the SMC instruction is UNDEFINED.



Encoding

SMC #<imm>

Decode for this encoding

```
// Empty.
```

Assembler symbols

<imm> Is a 16-bit unsigned immediate, in the range 0 to 65535, encoded in the "imm16" field.

Operation

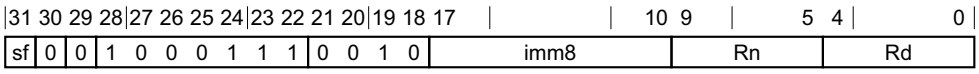
```
AArch64.CheckForSMCUndefOrTrap(imm16);
AArch64.CallSecureMonitor(imm16);
```

C6.2.324 SMIN (immediate)

Signed Minimum (immediate) determines the signed minimum of the source register value and immediate, and writes the result to the destination register.

Integer

(FEAT_CSSC)



32-bit variant

Applies when `sf == 0`.

```
SMIN <Wd>, <Wn>, #<simm>
```

64-bit variant

Applies when `sf == 1`.

```
SMIN <Xd>, <Xn>, #<simm>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_CSSC) then UNDEFINED;
constant integer datasize = 32 << UInt(sf);
integer n = UInt(Rn);
integer d = UInt(Rd);
integer imm = SInt(imm8);
```

Assembler symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
<simm>	Is a signed immediate, in the range -128 to 127, encoded in the "imm8" field.

Operation

```
bits(datasize) operand1 = X[n, datasize];
integer result = Min(SInt(operand1), imm);
X[d, datasize] = result<datasize-1:0>;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

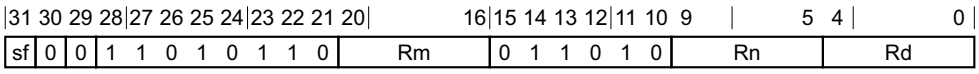
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

C6.2.325 SMIN (register)

Signed Minimum (register) determines the signed minimum of the two source register values and writes the result to the destination register.

Integer

(FEAT_CSSC)



32-bit variant

Applies when `sf == 0`.

```
SMIN <Wd>, <Wn>, <Wm>
```

64-bit variant

Applies when `sf == 1`.

```
SMIN <Xd>, <Xn>, <Xm>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_CSSC) then UNDEFINED;
constant integer datasize = 32 << UInt(sf);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer d = UInt(Rd);
```

Assembler symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Wm>	Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.

Operation

```
bits(datasize) operand1 = X[n, datasize];
bits(datasize) operand2 = X[m, datasize];
integer result = Min(SInt(operand1), SInt(operand2));
X[d, datasize] = result<datasize-1:0>;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.

  - The values of the NZCV flags.

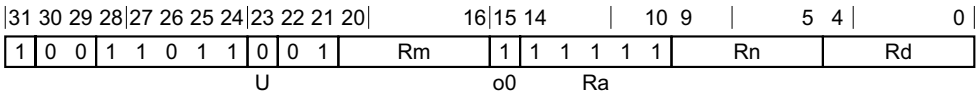
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

C6.2.326 SMNEGL

Signed Multiply-Negate Long multiplies two 32-bit register values, negates the product, and writes the result to the 64-bit destination register.

This instruction is an alias of the SMSUBL instruction. This means that:

- The encodings in this description are named to match the encodings of SMSUBL.
- The description of SMSUBL gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.



Encoding

```
SMNEGL <Xd>, <Wn>, <Wm>
```

is equivalent to

```
SMSUBL <Xd>, <Wn>, <Wm>, XZR
```

and is always the preferred disassembly.

Assembler symbols

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Wn> Is the 32-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.

<Wm> Is the 32-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.

Operation

The description of SMSUBL gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

C6.2.327 SMSTART

Enables access to Streaming SVE mode and SME architectural state.

SMSTART enters Streaming SVE mode, and enables the SME ZA storage.

SMSTART SM enters Streaming SVE mode, but does not enable the SME ZA storage.

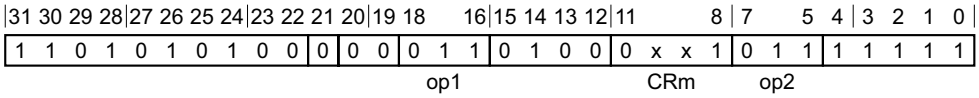
SMSTART ZA enables the SME ZA storage, but does not cause an entry to Streaming SVE mode.

This instruction is an alias of the MSR (immediate) instruction. This means that:

- The encodings in this description are named to match the encodings of MSR (immediate).
- The description of MSR (immediate) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

System

(FEAT_SME)



Encoding

`SMSTART {<option>}`

is equivalent to

`MSR  <pstatefield>, #1`

and is always the preferred disassembly.

Assembler symbols

`<option>` Is an optional mode, encoded in the "CRm<2:1>" field. It can have the following values:

`SM`	when `CRm<2:1> = 01`
`ZA`	when `CRm<2:1> = 10`
`[no specifier]`	when `CRm<2:1> = 11`

The encoding `CRm<2:1> = 00` is reserved.

`<pstatefield>` Is a PSTATE field name. For the MSR instruction, this is encoded in the "op1:op2:CRm" field. It can have the following values:

`SPSel`	when `op1 = 000`, `op2 = 101`, `CRm = xxxx`
`DAIFSet`	when `op1 = 011`, `op2 = 110`, `CRm = xxxx`
`DAIFClr`	when `op1 = 011`, `op2 = 111`, `CRm = xxxx`

When FEAT_UAO is implemented, the following value is also valid:

`UAO`	when `op1 = 000`, `op2 = 011`, `CRm = xxxx`

When FEAT_PAN is implemented, the following value is also valid:

`PAN`	when `op1 = 000`, `op2 = 100`, `CRm = xxxx`

When FEAT_NMI is implemented, the following value is also valid:

`ALLINT`	when `op1 = 001`, `op2 = 000`, `CRm = 000x`

When FEAT_EBEP is implemented, the following value is also valid:

`PM`	when `op1 = 001`, `op2 = 000`, `CRm = 001x`

When FEAT_SSBS is implemented, the following value is also valid:

SSBS when op1 = 011, op2 = 001, CRm = xxxx

When FEAT_DIT is implemented, the following value is also valid:

DIT when op1 = 011, op2 = 010, CRm = xxxx

When FEAT_SME is implemented, the following values are also valid:

SVCRSM when op1 = 011, op2 = 011, CRm = 001x

SVCRZA when op1 = 011, op2 = 011, CRm = 010x

SVCRSMZA when op1 = 011, op2 = 011, CRm = 011x

When FEAT_MTE is implemented, the following value is also valid:

TCO when op1 = 011, op2 = 100, CRm = xxxx

See *PSTATE* when op1 = 000, op2 = 00x, CRm = xxxx.

See *PSTATE* when op1 = 000, op2 = 010, CRm = xxxx.

The following encodings are reserved:

- op1 = 000, op2 = 11x, CRm = xxxx.
- op1 = 001, op2 = 000, CRm = 01xx.
- op1 = 001, op2 = 000, CRm = 1xxx.
- op1 = 001, op2 = 001, CRm = xxxx.
- op1 = 001, op2 = 01x, CRm = xxxx.
- op1 = 001, op2 = 1xx, CRm = xxxx.
- op1 = 010, op2 = xxx, CRm = xxxx.
- op1 = 011, op2 = 000, CRm = xxxx.
- op1 = 011, op2 = 011, CRm = 000x.
- op1 = 011, op2 = 011, CRm = 1xxx.
- op1 = 011, op2 = 101, CRm = xxxx.
- op1 = 1xx, op2 = xxx, CRm = xxxx.

For the SMSTART and SMSTOP aliases, this is encoded in "CRm<2:1>", where 0b01 specifies SVCRSM, 0b10 specifies SVCRZA, and 0b11 specifies SVCRSMZA.

Operation

The description of MSR (immediate) gives the operational pseudocode for this instruction.

C6.2.328 SMSTOP

Disables access to Streaming SVE mode and SME architectural state.

SMSTOP exits Streaming SVE mode, and disables the SME ZA storage.

SMSTOP SM exits Streaming SVE mode, but does not disable the SME ZA storage.

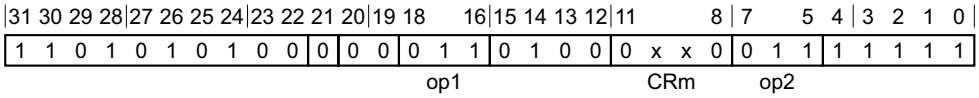
SMSTOP ZA disables the SME ZA storage, but does not cause an exit from Streaming SVE mode.

This instruction is an alias of the MSR (immediate) instruction. This means that:

- The encodings in this description are named to match the encodings of MSR (immediate).
- The description of MSR (immediate) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

System

(FEAT_SME)



Encoding

```
SMSTOP {<option>}
```

is equivalent to

```
MSR  <pstatefield>, #0
```

and is always the preferred disassembly.

Assembler symbols

`<option>` Is an optional mode, encoded in the "CRm<2:1>" field. It can have the following values:

`SM`	when `CRm<2:1> = 01`
`ZA`	when `CRm<2:1> = 10`
`[no specifier]`	when `CRm<2:1> = 11`

The encoding `CRm<2:1> = 00` is reserved.

`<pstatefield>` Is a PSTATE field name. For the MSR instruction, this is encoded in the "op1:op2:CRm" field. It can have the following values:

`SPSel`	when `op1 = 000, op2 = 101, CRm = xxxx`
`DAIFSet`	when `op1 = 011, op2 = 110, CRm = xxxx`
`DAIFClr`	when `op1 = 011, op2 = 111, CRm = xxxx`

When FEAT_UAO is implemented, the following value is also valid:

`UAO`	when `op1 = 000, op2 = 011, CRm = xxxx`

When FEAT_PAN is implemented, the following value is also valid:

`PAN`	when `op1 = 000, op2 = 100, CRm = xxxx`

When FEAT_NMI is implemented, the following value is also valid:

`ALLINT`	when `op1 = 001, op2 = 000, CRm = 000x`

When FEAT_EBEP is implemented, the following value is also valid:

`PM`	when `op1 = 001, op2 = 000, CRm = 001x`

When FEAT_SSBS is implemented, the following value is also valid:

SSBS when op1 = 011, op2 = 001, CRm = xxxx

When FEAT_DIT is implemented, the following value is also valid:

DIT when op1 = 011, op2 = 010, CRm = xxxx

When FEAT_SME is implemented, the following values are also valid:

SVCRSM when op1 = 011, op2 = 011, CRm = 001x

SVCRZA when op1 = 011, op2 = 011, CRm = 010x

SVCRSMZA when op1 = 011, op2 = 011, CRm = 011x

When FEAT_MTE is implemented, the following value is also valid:

TCO when op1 = 011, op2 = 100, CRm = xxxx

See *PSTATE* when op1 = 000, op2 = 00x, CRm = xxxx.

See *PSTATE* when op1 = 000, op2 = 010, CRm = xxxx.

The following encodings are reserved:

- op1 = 000, op2 = 11x, CRm = xxxx.
- op1 = 001, op2 = 000, CRm = 01xx.
- op1 = 001, op2 = 000, CRm = 1xxx.
- op1 = 001, op2 = 001, CRm = xxxx.
- op1 = 001, op2 = 01x, CRm = xxxx.
- op1 = 001, op2 = 1xx, CRm = xxxx.
- op1 = 010, op2 = xxx, CRm = xxxx.
- op1 = 011, op2 = 000, CRm = xxxx.
- op1 = 011, op2 = 011, CRm = 000x.
- op1 = 011, op2 = 011, CRm = 1xxx.
- op1 = 011, op2 = 101, CRm = xxxx.
- op1 = 1xx, op2 = xxx, CRm = xxxx.

For the SMSTART and SMSTOP aliases, this is encoded in "CRm<2:1>", where 0b01 specifies SVCRSM, 0b10 specifies SVCRZA, and 0b11 specifies SVCRSMZA.

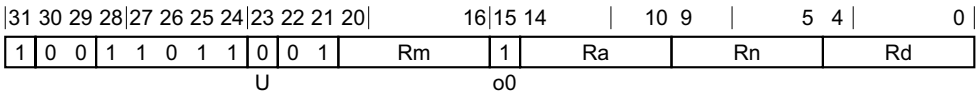
Operation

The description of MSR (immediate) gives the operational pseudocode for this instruction.

C6.2.329 SMSUBL

Signed Multiply-Subtract Long multiplies two 32-bit register values, subtracts the product from a 64-bit register value, and writes the result to the 64-bit destination register.

This instruction is used by the alias SMNEGL. See *Alias conditions* for details of when each alias is preferred.



Encoding

SMSUBL <Xd>, <Wn>, <Wm>, <Xa>

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer a = UInt(Ra);
```

Alias conditions

Alias	is preferred when
SMNEGL	Ra == '11111'

Assembler symbols

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Wn> Is the 32-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.

<Wm> Is the 32-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.

<Xa> Is the 64-bit name of the third general-purpose source register holding the minuend, encoded in the "Ra" field.

Operation

```
bits(32) operand1 = X[n, 32];
bits(32) operand2 = X[m, 32];
bits(64) operand3 = X[a, 64];

integer result;

result = Int(operand3, FALSE) - (Int(operand1, FALSE) * Int(operand2, FALSE));
X[d, 64] = result<63:0>;
```

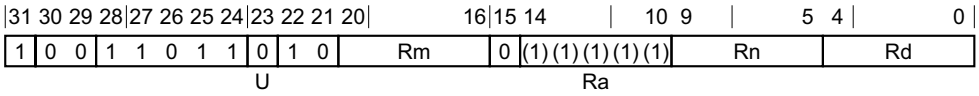
Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

C6.2.330 SMULH

Signed Multiply High multiplies two 64-bit register values, and writes bits[127:64] of the 128-bit result to the 64-bit destination register.



Encoding

```
SMULH <Xd>, <Xn>, <Xm>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
```

Assembler symbols

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Xn> Is the 64-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.

<Xm> Is the 64-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.

Operation

```
bits(64) operand1 = X[n, 64];
bits(64) operand2 = X[m, 64];

integer result;

result = Int(operand1, FALSE) * Int(operand2, FALSE);

X[d, 64] = result<127:64>;
```

Operational information

If PSTATE.DIT is 1:

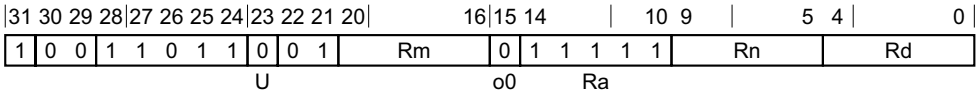
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

C6.2.331 SMULL

Signed Multiply Long multiplies two 32-bit register values, and writes the result to the 64-bit destination register.

This instruction is an alias of the SMADDL instruction. This means that:

- The encodings in this description are named to match the encodings of SMADDL.
- The description of SMADDL gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.



Encoding

```
SMULL <Xd>, <Wn>, <Wm>
```

is equivalent to

```
SMADDL <Xd>, <Wn>, <Wm>, XZR
```

and is always the preferred disassembly.

Assembler symbols

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Wn> Is the 32-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.

<Wm> Is the 32-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.

Operation

The description of SMADDL gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

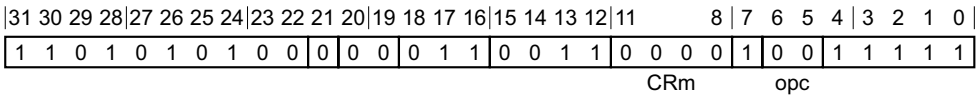
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

C6.2.332 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)*.

This instruction is an alias of the DSB instruction. This means that:

- The encodings in this description are named to match the encodings of DSB.
- The description of DSB gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.



Encoding

`SSBB`

is equivalent to

`DSB #0`

and is always the preferred disassembly.

Operation

The description of DSB gives the operational pseudocode for this instruction.

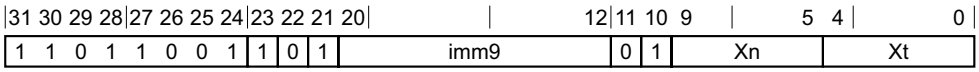
C6.2.333 ST2G

Store Allocation Tags stores an Allocation Tag to two Tag granules of memory. The address used for the store is calculated from the base register and an immediate signed offset scaled by the Tag granule. The Allocation Tag is calculated from the Logical Address Tag in the source register.

This instruction generates an Unchecked access.

Post-index

(FEAT_MTE)



Encoding

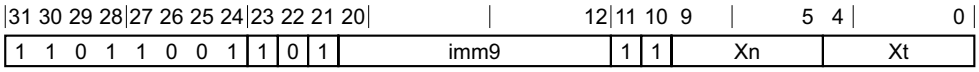
```
ST2G <Xt|SP>, [<Xn|SP>], #<simm>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_MTE) then UNDEFINED;
integer n = UInt(Xn);
integer t = UInt(Xt);
bits(64) offset = LSL(SignExtend(imm9, 64), LOG2_TAG_GRANULE);
boolean writeback = TRUE;
boolean postindex = TRUE;
```

Pre-index

(FEAT_MTE)



Encoding

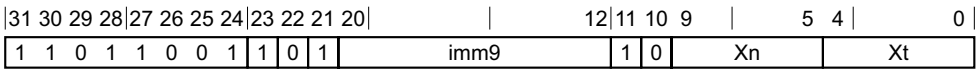
```
ST2G <Xt|SP>, [<Xn|SP>, #<simm>]!
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_MTE) then UNDEFINED;
integer n = UInt(Xn);
integer t = UInt(Xt);
bits(64) offset = LSL(SignExtend(imm9, 64), LOG2_TAG_GRANULE);
boolean writeback = TRUE;
boolean postindex = FALSE;
```

Signed offset

(FEAT_MTE)



Encoding

```
ST2G <Xt|SP>, [<Xn|SP>{, #<simm>}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_MTE) then UNDEFINED;
integer n = UInt(Xn);
integer t = UInt(Xt);
bits(64) offset = LSL(SignExtend(imm9, 64), LOG2_TAG_GRANULE);
boolean writeback = FALSE;
boolean postindex = FALSE;
```

Assembler symbols

<Xt|SP>	Is the 64-bit name of the general-purpose source register or stack pointer, encoded in the "Xt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Xn" field.
<simm>	Is the optional signed immediate offset, a multiple of 16 in the range -4096 to 4080, defaulting to 0 and encoded in the "imm9" field.

Operation for all encodings

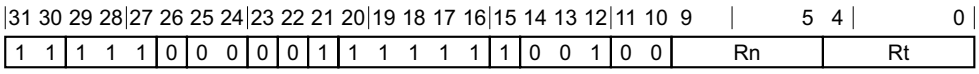
```
bits(64) address;
bits(64) address2;
bits(64) data = if t == 31 then SP[] else X[t, 64];
bits(4) tag = AArch64.AllocationTagFromAddress(data);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

AccessDescriptor accdesc = CreateAccDescLDGSTG(MemOp_STORE);

if !postindex then
    address = GenerateAddress(address, offset, accdesc);

address2 = GenerateAddress(address, TAG_GRANULE, accdesc);

AArch64.MemTag[address, accdesc] = tag;
AArch64.MemTag[address2, accdesc] = tag;

if writeback then
    if postindex then
        address = GenerateAddress(address, offset, accdesc);

    if n == 31 then
        SP[] = address;
    else
        X[n, 64] = address;
```

C6.2.334 ST64B

Single-copy Atomic 64-byte Store without status result stores eight 64-bit doublewords from consecutive registers, `Xt` to `X(t+7)`, to a memory location. The data that is stored is atomic and is required to be 64-byte aligned.

Integer

(FEAT_LS64)



Encoding

```
ST64B <Xt>, [<Xn|SP> {,#0}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_LS64) then UNDEFINED;
if Rt<4:3> == '11' || Rt<0> == '1' then UNDEFINED;

integer n = UInt(Rn);
integer t = UInt(Rt);
MemOp memop = MemOp_STORE;
boolean tagchecked = n != 31;
```

Assembler symbols

<Xt>	Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

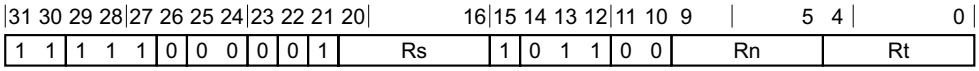
```
CheckLDST64BEnabled();

bits(512) data;
bits(64) address;
bits(64) value;

AccessDescriptor accdesc = CreateAccDescLS64(memop, tagchecked);
for i = 0 to 7
    value = X[t+i, 64];
    if BigEndian(accdesc.acctype) then value = BigEndianReverse(value);
    data<63+64*i:64*i> = value;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

MemStore64B(address, data, accdesc);
```

C6.2.335 ST64BV

Single-copy Atomic 64-byte Store with status result stores eight 64-bit doublewords from consecutive registers, `Xt` to `X(t+7)`, to a memory location, and writes the status result of the store to a register. The data that is stored is atomic and is required to be 64-byte aligned.

Integer

(FEAT_LS64_V)



Encoding

```
ST64BV <Xs>, <Xt>, [<Xn|SP>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_LS64_V) then UNDEFINED;
if Rt<4:3> == '11' || Rt<0> == '1' then UNDEFINED;

integer n = UInt(Rn);
integer t = UInt(Rt);
MemOp memop = MemOp_STORE;
integer s = UInt(Rs);
boolean tagchecked = n != 31;
```

Assembler symbols

`<Xs>` Is the 64-bit name of the general-purpose register into which the status result of this instruction is written, encoded in the "Rs" field.

The value returned is:

`0xFFFFFFFF_FFFFFFFF` If the memory location accessed does not support this instruction. In this case, the value at the memory location is UNKNOWN.

`!= 0xFFFFFFFF_FFFFFFFF` If the memory location accessed does support this instruction. In this case, the peripheral that provides the response defines the returned value and provides information on the state of the memory update at the memory location.

If XZR is used, then the return value is ignored.

`<Xt>` Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.

`<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
CheckST64BVEnabled();

bits(512) data;
bits(64) address;
bits(64) value;
bits(64) status;

AccessDescriptor accdesc = CreateAccDescLS64(memop, tagchecked);
for i = 0 to 7
    value = X[t+i, 64];
    if BigEndian(accdesc.acctype) then value = BigEndianReverse(value);
    data<63+64*i:64*i> = value;
```

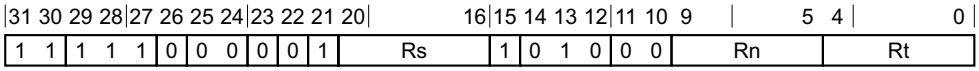
```
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

status = MemStore64BWithRet(address, data, accdesc);

if s != 31 then X[s, 64] = status;
```

C6.2.336 ST64BV0

Single-copy Atomic 64-byte EL0 Store with status result stores eight 64-bit doublewords from consecutive registers, `Xt` to `X(t+7)`, to a memory location, with the bottom 32 bits taken from ACCDATA_EL1, and writes the status result of the store to a register. The data that is stored is atomic and is required to be 64-byte aligned.

Integer

(FEAT_LS64_ACCDATA)



Encoding

```
ST64BV0 <Xs>, <Xt>, [<Xn|SP>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_LS64_ACCDATA) then UNDEFINED;
if Rt<4:3> == '11' || Rt<0> == '1' then UNDEFINED;

integer n = UInt(Rn);
integer t = UInt(Rt);
MemOp memop = MemOp_STORE;
integer s = UInt(Rs);
boolean tagchecked = n != 31;
```

Assembler symbols

`<Xs>` Is the 64-bit name of the general-purpose register into which the status result of this instruction is written, encoded in the "Rs" field.

The value returned is:

`0xFFFFFFFF_FFFFFFFF` If the memory location accessed does not support this instruction. In this case, the value at the memory location is UNKNOWN.

`!= 0xFFFFFFFF_FFFFFFFF` If the memory location accessed does support this instruction. In this case, the peripheral that provides the response defines the returned value and provides information on the state of the memory update at the memory location.

If XZR is used, then the return value is ignored.

`<Xt>` Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.

`<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
CheckST64BV0Enabled();

bits(512) data;
bits(64) address;
bits(64) value;
bits(64) status;

AccessDescriptor accdesc = CreateAccDescLS64(memop, tagchecked);
bits(64) Xt = X[t, 64];
value<31:0> = ACCDATA_EL1<31:0>;
value<63:32> = Xt<63:32>;
if BigEndian(accdesc.acctype) then value = BigEndianReverse(value);
```

```
data<63:0> = value;
for i = 1 to 7
    value = X[t+i, 64];
    if BigEndian(accdesc.acctype) then value = BigEndianReverse(value);
    data<63+64*i:64*i> = value;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

status = MemStore64BWithRet(address, data, accdesc);

if s != 31 then X[s, 64] = status;
```

C6.2.337 STADD, STADDL

Atomic add on word or doubleword in memory, without return, atomically loads a 32-bit word or 64-bit doubleword from memory, adds the value held in a register to it, and stores the result back to memory.

- STADD does not have release semantics.
- STADDL stores to memory with release semantics, as described in *Load-Acquire, Load-AcquirePC, and Store-Release*.

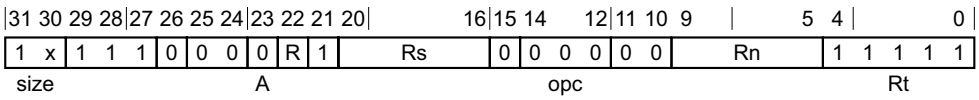
For information about memory accesses, see *Load/store addressing modes*.

This instruction is an alias of the LDADD, LDADDA, LDADDAL, LDADDL instruction. This means that:

- The encodings in this description are named to match the encodings of LDADD, LDADDA, LDADDAL, LDADDL.
- The description of LDADD, LDADDA, LDADDAL, LDADDL gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Integer

(FEAT_LSE)



32-bit LDADD alias variant

Applies when `size == 10 && R == 0`.

```
STADD <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDADD <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

32-bit LDADDL alias variant

Applies when `size == 10 && R == 1`.

```
STADDL <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDADDL <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

64-bit LDADD alias variant

Applies when `size == 11 && R == 0`.

```
STADD <Xs>, [<Xn|SP>]
```

is equivalent to

```
LDADD <Xs>, XZR, [<Xn|SP>]
```

and is always the preferred disassembly.

64-bit LDADDL alias variant

Applies when `size == 11 && R == 1`.

```
STADDL <Xs>, [<Xn|SP>]
```

is equivalent to

```
LDADDL <Xs>, XZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Assembler symbols

`<Ws>`	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
`<Xs>`	Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
`<Xn|SP>`	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

The description of LDADD, LDADDA, LDADDAL, LDADDL gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

C6.2.338 STADDB, STADDLB

Atomic add on byte in memory, without return, atomically loads an 8-bit byte from memory, adds the value held in a register to it, and stores the result back to memory.

- `STADDB` does not have release semantics.
- `STADDLB` stores to memory with release semantics, as described in *Load-Acquire, Load-AcquirePC, and Store-Release*.

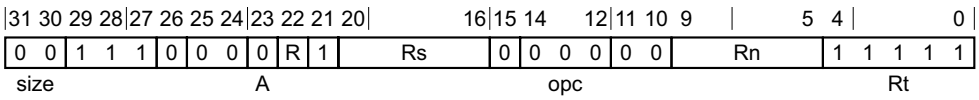
For information about memory accesses, see *Load/store addressing modes*.

This instruction is an alias of the LDADDB, LDADDAB, LDADDALB, LDADDLB instruction. This means that:

- The encodings in this description are named to match the encodings of LDADDB, LDADDAB, LDADDALB, LDADDLB.
- The description of LDADDB, LDADDAB, LDADDALB, LDADDLB gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Integer

(FEAT_LSE)



No memory ordering variant

Applies when `R == 0`.

```
STADDB <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDADDB <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Release variant

Applies when `R == 1`.

```
STADDLB <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDADDLB <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Assembler symbols

`<Ws>`	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
`<Xn|SP>`	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

The description of LDADDB, LDADDAB, LDADDALB, LDADDLB gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

C6.2.339 STADDH, STADDLH

Atomic add on halfword in memory, without return, atomically loads a 16-bit halfword from memory, adds the value held in a register to it, and stores the result back to memory.

- `STADDH` does not have release semantics.
- `STADDLH` stores to memory with release semantics, as described in *Load-Acquire, Load-AcquirePC, and Store-Release*.

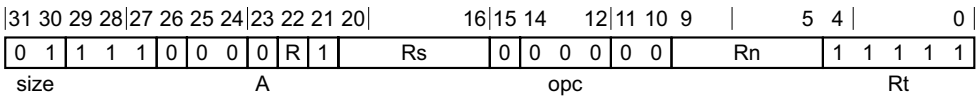
For information about memory accesses, see *Load/store addressing modes*.

This instruction is an alias of the LDADDH, LDADDAH, LDADDALH, LDADDLH instruction. This means that:

- The encodings in this description are named to match the encodings of LDADDH, LDADDAH, LDADDALH, LDADDLH.
- The description of LDADDH, LDADDAH, LDADDALH, LDADDLH gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Integer

(FEAT_LSE)



No memory ordering variant

Applies when `R == 0`.

```
STADDH <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDADDH <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Release variant

Applies when `R == 1`.

```
STADDLH <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDADDLH <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Assembler symbols

`<Ws>`	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
`<Xn|SP>`	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

The description of LDADDH, LDADDAH, LDADDALH, LDADDLH gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

C6.2.340 STCLR, STCLRL

Atomic bit clear on word or doubleword in memory, without return, atomically loads a 32-bit word or 64-bit doubleword from memory, performs a bitwise AND with the complement of the value held in a register on it, and stores the result back to memory.

- STCLR does not have release semantics.
- STCLRL stores to memory with release semantics, as described in *Load-Acquire, Load-AcquirePC, and Store-Release*.

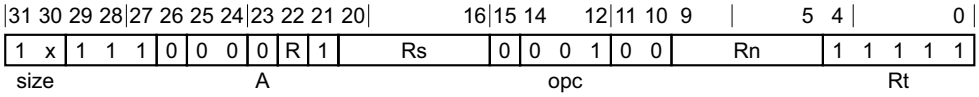
For information about memory accesses, see *Load/store addressing modes*.

This instruction is an alias of the LDCLR, LDCLRA, LDCLRAL, LDCLRL instruction. This means that:

- The encodings in this description are named to match the encodings of LDCLR, LDCLRA, LDCLRAL, LDCLRL.
- The description of LDCLR, LDCLRA, LDCLRAL, LDCLRL gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Integer

(FEAT_LSE)



32-bit LDCLR alias variant

Applies when `size == 10 && R == 0`.

```
STCLR <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDCLR <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

32-bit LDCLRL alias variant

Applies when `size == 10 && R == 1`.

```
STCLRL <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDCLRL <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

64-bit LDCLR alias variant

Applies when `size == 11 && R == 0`.

```
STCLR <Xs>, [<Xn|SP>]
```

is equivalent to

```
LDCLR <Xs>, XZR, [<Xn|SP>]
```

and is always the preferred disassembly.

64-bit LDCLRL alias variant

Applies when `size == 11 && R == 1`.

```
STCLRL <Xs>, [<Xn|SP>]
```

is equivalent to

```
LDCLRL <Xs>, XZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Assembler symbols

`<Ws>`	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
`<Xs>`	Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
`<Xn|SP>`	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

The description of LDCLR, LDCLRA, LDCLRAL, LDCLRL gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

C6.2.341 STCLRB, STCLRLB

Atomic bit clear on byte in memory, without return, atomically loads an 8-bit byte from memory, performs a bitwise AND with the complement of the value held in a register on it, and stores the result back to memory.

- `STCLRB` does not have release semantics.
- `STCLRLB` stores to memory with release semantics, as described in *Load-Acquire, Load-AcquirePC, and Store-Release*.

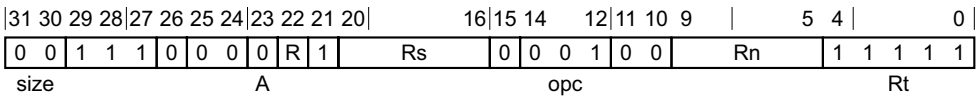
For information about memory accesses, see *Load/store addressing modes*.

This instruction is an alias of the LDCLRB, LDCLRAB, LDCLRALB, LDCLRLB instruction. This means that:

- The encodings in this description are named to match the encodings of LDCLRB, LDCLRAB, LDCLRALB, LDCLRLB.
- The description of LDCLRB, LDCLRAB, LDCLRALB, LDCLRLB gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Integer

(FEAT_LSE)



No memory ordering variant

Applies when `R == 0`.

```
STCLRB <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDCLRB <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Release variant

Applies when `R == 1`.

```
STCLRLB <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDCLRLB <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Assembler symbols

`<Ws>`	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
`<Xn|SP>`	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

The description of LDCLRB, LDCLRAB, LDCLRALB, LDCLRLB gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

C6.2.342 STCLRH, STCLRLH

Atomic bit clear on halfword in memory, without return, atomically loads a 16-bit halfword from memory, performs a bitwise AND with the complement of the value held in a register on it, and stores the result back to memory.

- STCLRH does not have release semantics.
- STCLRLH stores to memory with release semantics, as described in *Load-Acquire, Load-AcquirePC, and Store-Release*.

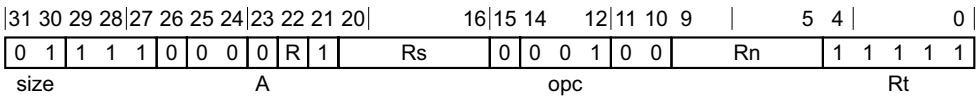
For information about memory accesses, see *Load/store addressing modes*.

This instruction is an alias of the LDCLRH, LDCLRAH, LDCLRALH, LDCLRLH instruction. This means that:

- The encodings in this description are named to match the encodings of LDCLRH, LDCLRAH, LDCLRALH, LDCLRLH.
- The description of LDCLRH, LDCLRAH, LDCLRALH, LDCLRLH gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Integer

(FEAT_LSE)



No memory ordering variant

Applies when R == 0.

```
STCLRH <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDCLRH <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Release variant

Applies when R == 1.

```
STCLRLH <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDCLRLH <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Assembler symbols

<Ws>	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

The description of LDCLRH, LDCLRAH, LDCLRALH, LDCLRLH gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

C6.2.343 STEOR, STEORL

Atomic Exclusive-OR on word or doubleword in memory, without return, atomically loads a 32-bit word or 64-bit doubleword from memory, performs an exclusive-OR with the value held in a register on it, and stores the result back to memory.

- `STEOR` does not have release semantics.
- `STEORL` stores to memory with release semantics, as described in *Load-Acquire, Load-AcquirePC, and Store-Release*.

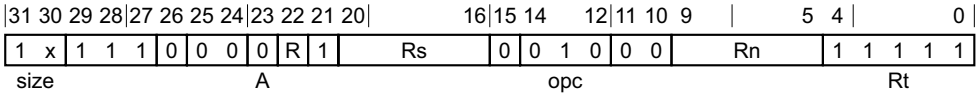
For information about memory accesses, see *Load/store addressing modes*.

This instruction is an alias of the LDEOR, LDEORA, LDEORAL, LDEORL instruction. This means that:

- The encodings in this description are named to match the encodings of LDEOR, LDEORA, LDEORAL, LDEORL.
- The description of LDEOR, LDEORA, LDEORAL, LDEORL gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Integer

(FEAT_LSE)



32-bit LDEOR alias variant

Applies when `size == 10 && R == 0`.

```
STEOR <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDEOR <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

32-bit LDEORL alias variant

Applies when `size == 10 && R == 1`.

```
STEORL <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDEORL <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

64-bit LDEOR alias variant

Applies when `size == 11 && R == 0`.

```
STEOR <Xs>, [<Xn|SP>]
```

is equivalent to

```
LDEOR <Xs>, XZR, [<Xn|SP>]
```

and is always the preferred disassembly.

64-bit LDEORL alias variant

Applies when `size == 11 && R == 1`.

```
STEORL <Xs>, [<Xn|SP>]
```

is equivalent to

```
LDEORL <Xs>, XZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Assembler symbols

`<Ws>`	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
`<Xs>`	Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
`<Xn|SP>`	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

The description of LDEOR, LDEORA, LDEORAL, LDEORL gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

C6.2.344 STEORB, STEORLB

Atomic Exclusive-OR on byte in memory, without return, atomically loads an 8-bit byte from memory, performs an exclusive-OR with the value held in a register on it, and stores the result back to memory.

- `STEORB` does not have release semantics.
- `STEORLB` stores to memory with release semantics, as described in *Load-Acquire, Load-AcquirePC, and Store-Release*.

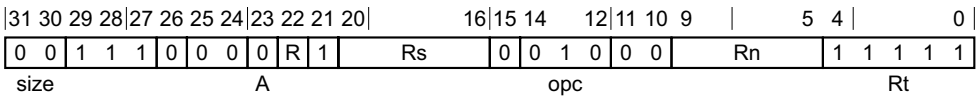
For information about memory accesses, see *Load/store addressing modes*.

This instruction is an alias of the LDEORB, LDEORAB, LDEORALB, LDEORLB instruction. This means that:

- The encodings in this description are named to match the encodings of LDEORB, LDEORAB, LDEORALB, LDEORLB.
- The description of LDEORB, LDEORAB, LDEORALB, LDEORLB gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Integer

(FEAT_LSE)



No memory ordering variant

Applies when `R == 0`.

```
STEORB <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDEORB <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Release variant

Applies when `R == 1`.

```
STEORLB <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDEORLB <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Assembler symbols

`<Ws>`	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
`<Xn|SP>`	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

The description of LDEORB, LDEORAB, LDEORALB, LDEORLB gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

C6.2.345 STEORH, STEORLH

Atomic Exclusive-OR on halfword in memory, without return, atomically loads a 16-bit halfword from memory, performs an exclusive-OR with the value held in a register on it, and stores the result back to memory.

- `STEORH` does not have release semantics.
- `STEORLH` stores to memory with release semantics, as described in *Load-Acquire, Load-AcquirePC, and Store-Release*.

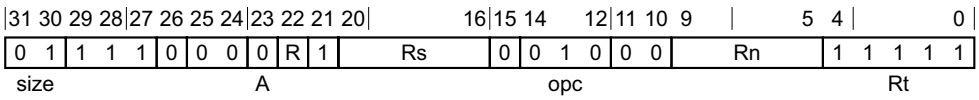
For information about memory accesses, see *Load/store addressing modes*.

This instruction is an alias of the LDEORH, LDEORAH, LDEORALH, LDEORLH instruction. This means that:

- The encodings in this description are named to match the encodings of LDEORH, LDEORAH, LDEORALH, LDEORLH.
- The description of LDEORH, LDEORAH, LDEORALH, LDEORLH gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Integer

(FEAT_LSE)



No memory ordering variant

Applies when `R == 0`.

```
STEORH <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDEORH <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Release variant

Applies when `R == 1`.

```
STEORLH <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDEORLH <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Assembler symbols

`<Ws>`	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
`<Xn|SP>`	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

The description of LDEORH, LDEORAH, LDEORALH, LDEORLH gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

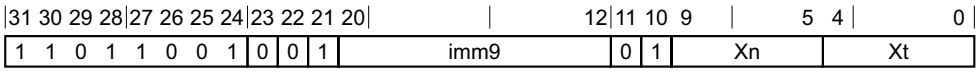
C6.2.346 STG

Store Allocation Tag stores an Allocation Tag to memory. The address used for the store is calculated from the base register and an immediate signed offset scaled by the Tag granule. The Allocation Tag is calculated from the Logical Address Tag in the source register.

This instruction generates an Unchecked access.

Post-index

(FEAT_MTE)



Encoding

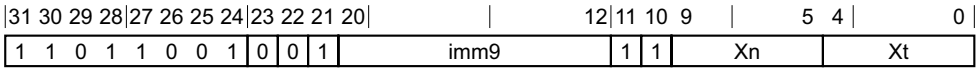
```
STG <Xt|SP>, [<Xn|SP>], #<simm>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_MTE) then UNDEFINED;
integer n = UInt(Xn);
integer t = UInt(Xt);
bits(64) offset = LSL(SignExtend(imm9, 64), LOG2_TAG_GRANULE);
boolean writeback = TRUE;
boolean postindex = TRUE;
```

Pre-index

(FEAT_MTE)



Encoding

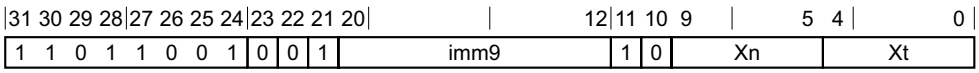
```
STG <Xt|SP>, [<Xn|SP>, #<simm>]!
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_MTE) then UNDEFINED;
integer n = UInt(Xn);
integer t = UInt(Xt);
bits(64) offset = LSL(SignExtend(imm9, 64), LOG2_TAG_GRANULE);
boolean writeback = TRUE;
boolean postindex = FALSE;
```

Signed offset

(FEAT_MTE)



Encoding

```
STG <Xt|SP>, [<Xn|SP>{, #<simm>}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_MTE) then UNDEFINED;
integer n = UInt(Xn);
integer t = UInt(Xt);
bits(64) offset = LSL(SignExtend(imm9, 64), LOG2_TAG_GRANULE);
boolean writeback = FALSE;
boolean postindex = FALSE;
```

Assembler symbols

<Xt|SP>	Is the 64-bit name of the general-purpose source register or stack pointer, encoded in the "Xt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Xn" field.
<simm>	Is the optional signed immediate offset, a multiple of 16 in the range -4096 to 4080, defaulting to 0 and encoded in the "imm9" field.

Operation for all encodings

```
bits(64) address;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

AccessDescriptor accdesc = CreateAccDescLDGSTG(MemOp_STORE);

if !postindex then
    address = GenerateAddress(address, offset, accdesc);

bits(64) data = if t == 31 then SP[] else X[t, 64];
bits(4) tag = AArch64.AllocationTagFromAddress(data);
AArch64.MemTag[address, accdesc] = tag;

if writeback then
    if postindex then
        address = GenerateAddress(address, offset, accdesc);

    if n == 31 then
        SP[] = address;
    else
        X[n, 64] = address;
```

C6.2.347 STGM

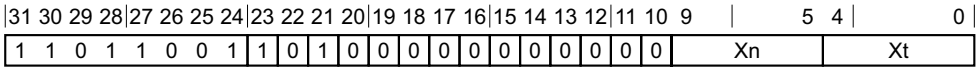
Store Tag Multiple writes a naturally aligned block of N Allocation Tags, where the size of N is identified in GMID_EL1.BS, and the Allocation Tag written to address A is taken from the source register at 4*A<7:4>+3:4*A<7:4>.

This instruction is UNDEFINED at EL0.

This instruction generates an Unchecked access.

Integer

(FEAT_MTE2)



Encoding

```
STGM <Xt>, [<Xn|SP>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_MTE2) then UNDEFINED;
integer t = UInt(Xt);
integer n = UInt(Xn);
```

Assembler symbols

<Xt> Is the 64-bit name of the general-purpose source register, encoded in the "Xt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Xn" field.

Operation

```
if PSTATE.EL == EL0 then
    UNDEFINED;

bits(64) data = X[t, 64];
bits(64) address;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

integer size = 4 * (2 ^ (UInt(GMID_EL1.BS)));
address = Align(address, size);
constant integer count = size >> LOG2_TAG_GRANULE;
integer index = UInt(address<LOG2_TAG_GRANULE+3:LOG2_TAG_GRANULE>);
constant bits(64) curraddress = address;
AccessDescriptor accdesc = CreateAccDescLDGSTG(MemOp_STORE);

for i = 0 to count-1
    bits(4) tag = Elem[data, index, 4];
    AArch64.MemTag[address, accdesc] = tag;
    address = GenerateAddress(address, TAG_GRANULE, accdesc);
    index = index + 1;
```

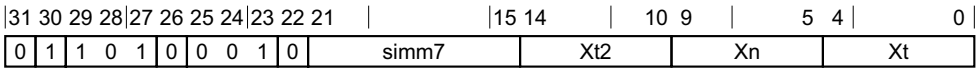
C6.2.348 STGP

Store Allocation Tag and Pair of registers stores an Allocation Tag and two 64-bit doublewords to memory, from two registers. The address used for the store is calculated from the base register and an immediate signed offset scaled by the Tag granule. The Allocation Tag is calculated from the Logical Address Tag in the base register.

This instruction generates an Unchecked access.

Post-index

(FEAT_MTE)



Encoding

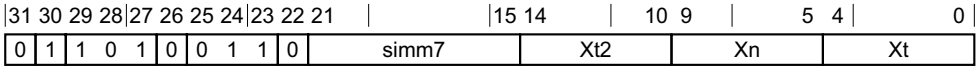
```
STGP <Xt1>, <Xt2>, [<Xn|SP>], #<imm>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_MTE) then UNDEFINED;
integer n = UInt(Xn);
integer t = UInt(Xt);
integer t2 = UInt(Xt2);
bits(64) offset = LSL(SignExtend(simm7, 64), LOG2_TAG_GRANULE);
boolean writeback = TRUE;
boolean postindex = TRUE;
```

Pre-index

(FEAT_MTE)



Encoding

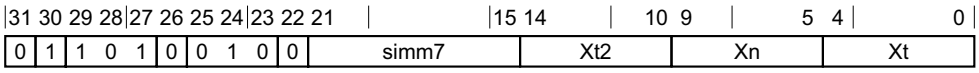
```
STGP <Xt1>, <Xt2>, [<Xn|SP>, #<imm>]!
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_MTE) then UNDEFINED;
integer n = UInt(Xn);
integer t = UInt(Xt);
integer t2 = UInt(Xt2);
bits(64) offset = LSL(SignExtend(simm7, 64), LOG2_TAG_GRANULE);
boolean writeback = TRUE;
boolean postindex = FALSE;
```

Signed offset

(FEAT_MTE)



Encoding

```
STGP <Xt1>, <Xt2>, [<Xn|SP>{, #<imm>}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_MTE) then UNDEFINED;
integer n = UInt(Xn);
integer t = UInt(Xt);
integer t2 = UInt(Xt2);
bits(64) offset = LSL(SignExtend(simm7, 64), LOG2_TAG_GRANULE);
boolean writeback = FALSE;
boolean postindex = FALSE;
```

Assembler symbols

<Xt1>	Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Xt" field.
<Xt2>	Is the 64-bit name of the second general-purpose register to be transferred, encoded in the "Xt2" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Xn" field.
<imm>	For the post-index and pre-index variant: is the signed immediate offset, a multiple of 16 in the range -1024 to 1008, encoded in the "simm7" field. For the signed offset variant: is the optional signed immediate offset, a multiple of 16 in the range -1024 to 1008, defaulting to 0 and encoded in the "simm7" field.

Operation for all encodings

```
bits(64) address;
bits(64) address2;
bits(64) data1;
bits(64) data2;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

data1 = X[t, 64];
data2 = X[t2, 64];

AccessDescriptor accdesc = CreateAccDescLDGSTG(MemOp_STORE);

if !postindex then
    address = GenerateAddress(address, offset, accdesc);

if !IsAligned(address, TAG_GRANULE) then
    AArch64.Abort(address, AlignmentFault(accdesc));

address2 = GenerateAddress(address, 8, accdesc);
Mem[address, 8, accdesc] = data1;
Mem[address2, 8, accdesc] = data2;

AArch64.MemTag[address, accdesc] = AArch64.AllocationTagFromAddress(address);
```

```
if writeback then
    if postindex then
        address = GenerateAddress(address, offset, accdesc);

    if n == 31 then
        SP[] = address;
    else
        X[n, 64] = address;
```

C6.2.349 STILP

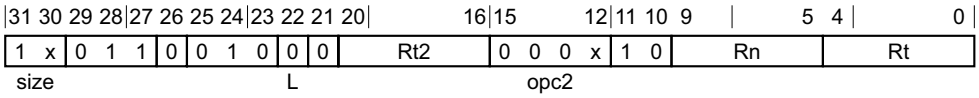
Store-Release ordered Pair of registers calculates an address from a base register value and an optional offset, and stores two 32-bit words or two 64-bit doublewords to the calculated address, from two registers. For information on single-copy atomicity and alignment requirements, see *Requirements for single-copy atomicity* and *Alignment of data accesses*. The instruction also has memory ordering semantics, as described in *Load-Acquire, Load-AcquirePC, and Store-Release*, with the additional requirement that:

- When using the pre-index addressing mode, the Memory effects associated with Xt2/Wt2 are Ordered-before the Memory effects associated with Xt1/Wt1.
- For all other addressing modes, the Memory effects associated with Xt1/Wt1 are Ordered-before the Memory effects associated with Xt2/Wt2.

For information about memory accesses, see *Load/store addressing modes*.

Integer

(FEAT_LRCPC3)



32-bit variant

Applies when `size == 10 && opc2 == 0001`.

```
STILP <Wt1>, <Wt2>, [<Xn|SP>]
```

32-bit pre-index variant

Applies when `size == 10 && opc2 == 0000`.

```
STILP <Wt1>, <Wt2>, [<Xn|SP>, #-8]!
```

64-bit variant

Applies when `size == 11 && opc2 == 0001`.

```
STILP <Xt1>, <Xt2>, [<Xn|SP>]
```

64-bit pre-index variant

Applies when `size == 11 && opc2 == 0000`.

```
STILP <Xt1>, <Xt2>, [<Xn|SP>, #-16]!
```

Decode for all variants of this encoding

```
boolean wback;
wback = opc2<0> == '0';
```

Notes for all encodings

STILP has the same CONSTRAINED UNPREDICTABLE behavior as STP. For information about this CONSTRAINED UNPREDICTABLE behavior, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*, and particularly *STP and STILP*.

Assembler symbols

<Wt1> Is the 32-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.

<Wt2> Is the 32-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.

<Xt1> Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.

<Xt2> Is the 64-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Shared decode for all encodings

```
integer offset;
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2);
constant integer scale = 2 + UInt(size<0>);
constant integer datasize = 8 << scale;
offset = if opc2<0> == '0' then -1 * (2 << scale) else 0;

boolean tagchecked = wback || n != 31;

boolean rt_unknown = FALSE;

if wback && (t == n || t2 == n) && n != 31 then
    Constraint c = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE    rt_unknown = FALSE;    // value stored is pre-writeback
        when Constraint_UNKNOWN rt_unknown = TRUE;    // value stored is UNKNOWN
        when Constraint_UNDEF   UNDEFINED;
        when Constraint_NOP     EndOfInstruction();
```

Operation

```
bits(64) address;
bits(64) address2;
bits(datasize) data1;
bits(datasize) data2;
constant integer dbytes = datasize DIV 8;

AccessDescriptor accdesc = CreateAccDescAcqRel(MemOp_STORE, tagchecked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

address = GenerateAddress(address, offset, accdesc);

if rt_unknown && t == n then
    data1 = bits(datasize) UNKNOWN;
else
    data1 = X[t, datasize];
if rt_unknown && t2 == n then
    data2 = bits(datasize) UNKNOWN;
else
    data2 = X[t2, datasize];

if IsFeatureImplemented(FEAT_LSE2) then
```

```
    bits(2*datasize) full_data;
    if BigEndian(accdesc.acctype) then
        full_data = data1:data2;
    else
        full_data = data2:data1;
    accdesc.ispair = TRUE;
    accdesc.highestaddressfirst = offset < 0;
    Mem[address, 2*dbytes, accdesc] = full_data;
else
    address2 = GenerateAddress(address, dbytes, accdesc);
    if offset < 0 then
        // Reverse the memory write order for negative pre-index.
        Mem[address2, dbytes, accdesc] = data2;
        Mem[address, dbytes, accdesc] = data1;
    else
        Mem[address, dbytes, accdesc] = data1;
        Mem[address2, dbytes, accdesc] = data2;
if wback then
    if n == 31 then
        SP[] = address;
    else
        X[n, 64] = address;
```

Operational information

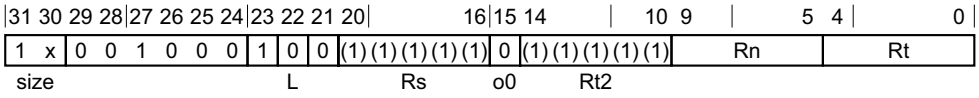
If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

C6.2.350 STLLR

Store LORelease Register stores a 32-bit word or a 64-bit doubleword to a memory location, from a register. The instruction also has memory ordering semantics as described in *LoadLOAcquire, StoreLORelease*. For information about memory accesses, see *Load/store addressing modes*.

No offset

(FEAT_LOR)



32-bit variant

Applies when `size == 10`.

```
STLLR <Wt>, [<Xn|SP>{,#0}]
```

64-bit variant

Applies when `size == 11`.

```
STLLR <Xt>, [<Xn|SP>{,#0}]
```

Decode for all variants of this encoding

```
integer n = UInt(Rn);
integer t = UInt(Rt);

constant integer elsize = 8 << UInt(size);
boolean tagchecked = n != 31;
```

Assembler symbols

`<Wt>`	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
`<Xt>`	Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
`<Xn|SP>`	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(elsize) data;
constant integer dbytes = elsize DIV 8;

AccessDescriptor accdesc;
accdesc = CreateAccDescLOR(MemOp_STORE, tagchecked);
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

data = X[t, elsize];
Mem[address, dbytes, accdesc] = data;
```

Operational information

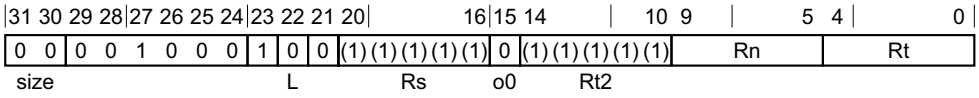
If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

C6.2.351 STLLRB

Store LORelease Register Byte stores a byte from a 32-bit register to a memory location. The instruction also has memory ordering semantics as described in *LoadLOAcquire, StoreLORelease*. For information about memory accesses, see *Load/store addressing modes*.

No offset

(FEAT_LOR)



Encoding

```
STLLRB <Wt>, [<Xn|SP>{,#0}]
```

Decode for this encoding

```
integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tagchecked = n != 31;
```

Assembler symbols

<Wt>	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(8) data;

AccessDescriptor accdesc;
accdesc = CreateAccDescLOR(MemOp_STORE, tagchecked);
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

data = X[t, 8];
Mem[address, 1, accdesc] = data;
```

Operational information

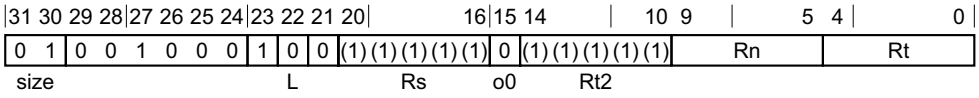
If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

C6.2.352 STLLRH

Store LORelease Register Halfword stores a halfword from a 32-bit register to a memory location. The instruction also has memory ordering semantics as described in *LoadLOAcquire, StoreLORelease*. For information about memory accesses, see *Load/store addressing modes*.

No offset

(FEAT_LOR)



Encoding

```
STLLRH <Wt>, [<Xn|SP>{,#0}]
```

Decode for this encoding

```
integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tagchecked = n != 31;
```

Assembler symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(16) data;

AccessDescriptor accdesc;
accdesc = CreateAccDescLOR(MemOp_STORE, tagchecked);
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

data = X[t, 16];
Mem[address, 2, accdesc] = data;
```

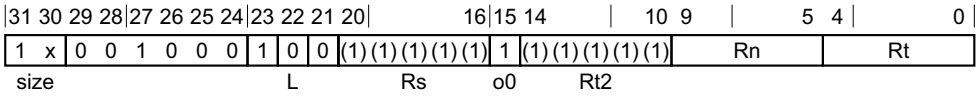
Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

C6.2.353 STLR

Store-Release Register stores a 32-bit word or a 64-bit doubleword to a memory location, from a register. The instruction also has memory ordering semantics as described in *Load-Acquire, Load-AcquirePC, and Store-Release*. For information about memory accesses, see *Load/store addressing modes*.

No offset



32-bit variant

Applies when `size == 10`.

```
STLR <Wt>, [<Xn|SP>{,#0}]
```

64-bit variant

Applies when `size == 11`.

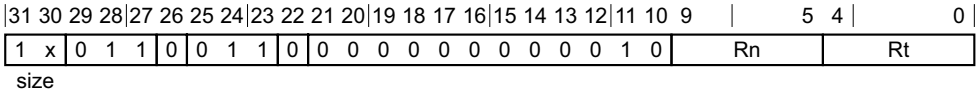
```
STLR <Xt>, [<Xn|SP>{,#0}]
```

Decode for all variants of this encoding

```
integer n = UInt(Rn);
integer t = UInt(Rt);
boolean wback = FALSE;
integer offset = 0;
boolean rt_unknown = FALSE;

constant integer elsize = 8 << UInt(size);
constant integer datasize = elsize;
boolean tagchecked = n != 31;
```

Pre-index

(FEAT_LRCPC3)



32-bit variant

Applies when `size == 10`.

```
STLR <Wt>, [<Xn|SP>, #-4]!
```

64-bit variant

Applies when `size == 11`.

```
STLR <Xt>, [<Xn|SP>, #-8]!
```

Decode for all variants of this encoding

```
boolean wback = TRUE;

integer n = UInt(Rn);
```

```
integer t = UInt(Rt);

constant integer datasize = 8 << UInt(size);
integer offset = -1 * (1 << UInt(size));
boolean tagchecked = TRUE;

boolean rt_unknown = FALSE;

if n == t && n != 31 then
    Constraint c = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE    rt_unknown = FALSE;    // value stored is original value
        when Constraint_UNKNOWN rt_unknown = TRUE;    // value stored is UNKNOWN
        when Constraint_UNDEF   UNDEFINED;
        when Constraint_NOP     EndOfInstruction();
```

Assembler symbols

<Wt>	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xt>	Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation for all encodings

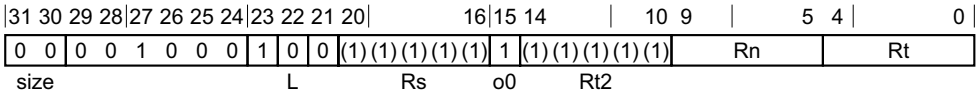
```
bits(64) address;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;

AccessDescriptor accdesc;
accdesc = CreateAccDescAcqRel(MemOp_STORE, tagchecked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

address = GenerateAddress(address, offset, accdesc);
if rt_unknown then
    data = bits(datasize) UNKNOWN;
else
    data = X[t, datasize];
Mem[address, dbytes, accdesc] = data;

if wback then
    if n == 31 then
        SP[] = address;
    else
        X[n, 64] = address;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

C6.2.354 STLRB

Store-Release Register Byte stores a byte from a 32-bit register to a memory location. The instruction also has memory ordering semantics as described in *Load-Acquire, Load-AcquirePC, and Store-Release*. For information about memory accesses, see *Load/store addressing modes*.



Encoding

```
STLRB <Wt>, [<Xn|SP>{,#0}]
```

Decode for this encoding

```
integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tagchecked = n != 31;
```

Assembler symbols

<Wt>	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

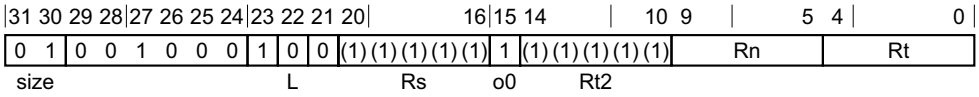
```
bits(64) address;
bits(8) data;

AccessDescriptor accdesc;
accdesc = CreateAccDescAcqRel(MemOp_STORE, tagchecked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

address = GenerateAddress(address, 0, accdesc);
data = X[t, 8];
Mem[address, 1, accdesc] = data;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

C6.2.355 STLRH

Store-Release Register Halfword stores a halfword from a 32-bit register to a memory location. The instruction also has memory ordering semantics as described in *Load-Acquire, Load-AcquirePC, and Store-Release*. For information about memory accesses, see *Load/store addressing modes*.



Encoding

```
STLRH <Wt>, [<Xn|SP>{,#0}]
```

Decode for this encoding

```
integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tagchecked = n != 31;
```

Assembler symbols

<Wt>	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(16) data;

AccessDescriptor accdesc;
accdesc = CreateAccDescAcqRel(MemOp_STORE, tagchecked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

address = GenerateAddress(address, 0, accdesc);
data = X[t, 16];
Mem[address, 2, accdesc] = data;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

C6.2.356 STLUR

Store-Release Register (unscaled) calculates an address from a base register value and an immediate offset, and stores a 32-bit word or a 64-bit doubleword to the calculated address, from a register.

The instruction has memory ordering semantics as described in *Load-Acquire, Load-AcquirePC, and Store-Release*

For information about memory accesses, see *Load/store addressing modes*.

Unscaled offset

(FEAT_LRCPC2)

32-bit variant

Applies when `size == 10`.

```
STLUR <Wt>, [<Xn|SP>{, #<simm>}]
```

64-bit variant

Applies when `size == 11`.

```
STLUR <Xt>, [<Xn|SP>{, #<simm>}]
```

Decode for all variants of this encoding

```
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

Assembler symbols

`<Wt>`	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
`<Xt>`	Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
`<Xn|SP>`	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
`<simm>`	Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared decode for all encodings

```
integer n = UInt(Rn);
integer t = UInt(Rt);

constant integer datasize = 8 << scale;
boolean tagchecked = n != 31;
```

Operation

```
bits(64) address;
bits(datasize) data;

AccessDescriptor accdesc;
accdesc = CreateAccDescAcqRel(MemOp_STORE, tagchecked);
```

```
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

address = GenerateAddress(address, offset, accdesc);

data = X[t, datasize];
Mem[address, datasize DIV 8, accdesc] = data;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

C6.2.357 STLURB

Store-Release Register Byte (unscaled) calculates an address from a base register value and an immediate offset, and stores a byte to the calculated address, from a 32-bit register.

The instruction has memory ordering semantics as described in *Load-Acquire, Load-AcquirePC, and Store-Release*

For information about memory accesses, see *Load/store addressing modes*.

Unscaled offset

(FEAT_LRCPC2)

Encoding

```
STLURB <Wt>, [<Xn|SP>{, #<simm>}]
```

Decode for this encoding

```
bits(64) offset = SignExtend(imm9, 64);
```

Assembler symbols

<Wt>	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<simm>	Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared decode for all encodings

```
integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tagchecked = n != 31;
```

Operation

```
bits(64) address;
bits(8) data;

AccessDescriptor accdesc;
accdesc = CreateAccDescAcqRel(MemOp_STORE, tagchecked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

address = GenerateAddress(address, offset, accdesc);

data = X[t, 8];
Mem[address, 1, accdesc] = data;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

C6.2.358 STLURH

Store-Release Register Halfword (unscaled) calculates an address from a base register value and an immediate offset, and stores a halfword to the calculated address, from a 32-bit register.

The instruction has memory ordering semantics as described in *Load-Acquire, Load-AcquirePC, and Store-Release*

For information about memory accesses, see *Load/store addressing modes*.

Unscaled offset

(FEAT_LRCPC2)

Encoding

```
STLURH <Wt>, [<Xn|SP>{, #<simm>}]
```

Decode for this encoding

```
bits(64) offset = SignExtend(imm9, 64);
```

Assembler symbols

<Wt>	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<simm>	Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared decode for all encodings

```
integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tagchecked = n != 31;
```

Operation

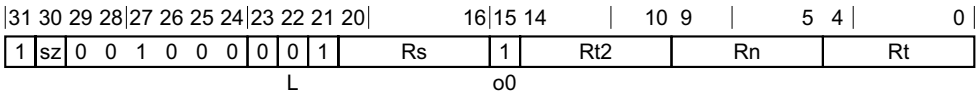
```
bits(64) address;
bits(16) data;

AccessDescriptor accdesc;
accdesc = CreateAccDescAcqRel(MemOp_STORE, tagchecked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

address = GenerateAddress(address, offset, accdesc);

data = X[t, 16];
Mem[address, 2, accdesc] = data;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

C6.2.359 STLXP

Store-Release Exclusive Pair of registers stores two 32-bit words or two 64-bit doublewords to a memory location if the PE has exclusive access to the memory address, from two registers, and returns a status value of 0 if the store was successful, or of 1 if no store was performed. See *Synchronization and semaphores*. For information on single-copy atomicity and alignment requirements, see *Requirements for single-copy atomicity* and *Alignment of data accesses*. If a 64-bit pair Store-Exclusive succeeds, it causes a single-copy atomic update of the 128-bit memory location being updated. The instruction also has memory ordering semantics, as described in *Load-Acquire, Load-AcquirePC, and Store-Release*. For information about memory accesses, see *Load/store addressing modes*.



32-bit variant

Applies when sz == 0.

```
STLXP <Ws>, <Wt1>, <Wt2>, [<Xn|SP>{,#0}]
```

64-bit variant

Applies when sz == 1.

```
STLXP <Ws>, <Xt1>, <Xt2>, [<Xn|SP>{,#0}]
```

Decode for all variants of this encoding

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2);    // ignored by load/store single register
integer s = UInt(Rs);    // ignored by all loads and store-release

constant integer elsize = 32 << UInt(sz);
constant integer datasize = elsize * 2;
boolean tagchecked = n != 31;

boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;
if s == t || (s == t2) then
    Constraint c = ConstrainUnpredictable(Unpredictable_DATAOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN rt_unknown = TRUE;    // store UNKNOWN value
        when Constraint_UNDEF   UNDEFINED;
        when Constraint_NOP     EndOfInstruction();
if s == n && n != 31 then
    Constraint c = ConstrainUnpredictable(Unpredictable_BASEOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN rn_unknown = TRUE;    // address is UNKNOWN
        when Constraint_UNDEF   UNDEFINED;
        when Constraint_NOP     EndOfInstruction();
```

Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*, and particularly *STLXP*.

Assembler symbols

<Ws> Is the 32-bit name of the general-purpose register into which the status result of the store exclusive is written, encoded in the "Rs" field. The value returned is:

0 If the operation updates memory.

1 If the operation fails to update memory.

<Xt1> Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.

<Xt2> Is the 64-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.

<Wt1> Is the 32-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.

<Wt2> Is the 32-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Aborts and alignment

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- <Ws> is not updated.

Accessing an address that is not aligned to the size of the data being accessed causes an Alignment fault Data Abort exception to be generated, subject to the following rules:

- If AArch64.ExclusiveMonitorsPass() returns TRUE, the exception is generated.
- Otherwise, it is IMPLEMENTATION DEFINED whether the exception is generated.

If AArch64.ExclusiveMonitorsPass() returns FALSE and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.

Operation

```
bits(64) address;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;

AccessDescriptor accdesc = CreateAccDescExLDST(MemOp_STORE, TRUE, tagchecked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
elsif rn_unknown then
    address = bits(64) UNKNOWN;
else
    address = X[n, 64];

if rt_unknown then
    data = bits(datasize) UNKNOWN;
else
    bits(datasize DIV 2) el1 = X[t, datasize DIV 2];
    bits(datasize DIV 2) el2 = X[t2, datasize DIV 2];
    data = if BigEndian(accdesc.acctype) then el1:el2 else el2:el1;
bit status = '1';
// Check whether the Exclusives monitors are set to include the
// physical memory locations corresponding to virtual address
// range [address, address+dbytes-1].

// If AArch64.ExclusiveMonitorsPass() returns FALSE and the memory address,
// if accessed, would generate a synchronous Data Abort exception, it is
```

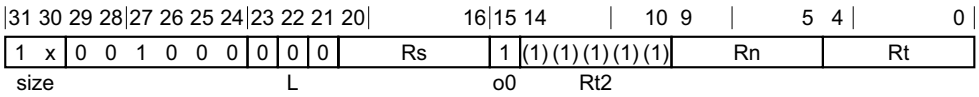
```
// IMPLEMENTATION DEFINED whether the exception is generated.
// It is a limitation of this model that synchronous Data Aborts are never
// generated in this case, as Mem[] is not called.
// If FEAT_SPE is implemented, it is also IMPLEMENTATION DEFINED whether or not the
// physical address packet is output when permitted and when
// AArch64.ExclusiveMonitorPass() returns FALSE for a Store Exclusive instruction.
// This behavior is not reflected here due to the previously stated limitation.
if AArch64.ExclusiveMonitorsPass(address, dbytes, accdesc) then
    // This atomic write will be rejected if it does not refer
    // to the same physical locations after address translation.
    Mem[address, dbytes, accdesc] = data;
    status = ExclusiveMonitorsStatus();
X[s, 32] = ZeroExtend(status, 32);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

C6.2.360 STLXR

Store-Release Exclusive Register stores a 32-bit word or a 64-bit doubleword to memory if the PE has exclusive access to the memory address, from two registers, and returns a status value of 0 if the store was successful, or of 1 if no store was performed. See *Synchronization and semaphores*. The memory access is atomic. The instruction also has memory ordering semantics as described in *Load-Acquire, Load-AcquirePC, and Store-Release*. For information about memory accesses, see *Load/store addressing modes*.



32-bit variant

Applies when size == 10.

```
STLXR <Ws>, <Wt>, [<Xn|SP>{,#0}]
```

64-bit variant

Applies when size == 11.

```
STLXR <Ws>, <Xt>, [<Xn|SP>{,#0}]
```

Decode for all variants of this encoding

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer s = UInt(Rs);    // ignored by all loads and store-release

constant integer elsize = 8 << UInt(size);
boolean tagchecked = n != 31;

boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;
if s == t then
    Constraint c = ConstrainUnpredictable(Unpredictable_DATAOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN rt_unknown = TRUE;    // store UNKNOWN value
        when Constraint_UNDEF   UNDEFINED;
        when Constraint_NOP     EndOfInstruction();
if s == n && n != 31 then
    Constraint c = ConstrainUnpredictable(Unpredictable_BASEOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN rn_unknown = TRUE;    // address is UNKNOWN
        when Constraint_UNDEF   UNDEFINED;
        when Constraint_NOP     EndOfInstruction();
```

Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*, and particularly *STLXR*.

Assembler symbols

<Ws> Is the 32-bit name of the general-purpose register into which the status result of the store exclusive is written, encoded in the "Rs" field. The value returned is:

0 If the operation updates memory.

1 If the operation fails to update memory.

<Xt> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Aborts and alignment

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- <Ws> is not updated.

Accessing an address that is not aligned to the size of the data being accessed causes an Alignment fault Data Abort exception to be generated, subject to the following rules:

- If AArch64.ExclusiveMonitorsPass() returns TRUE, the exception is generated.
- Otherwise, it is IMPLEMENTATION DEFINED whether the exception is generated.

If AArch64.ExclusiveMonitorsPass() returns FALSE and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.

Operation

```
bits(64) address;
bits(elsize) data;
constant integer dbytes = elsize DIV 8;

AccessDescriptor accdesc = CreateAccDescExLDST(MemOp_STORE, TRUE, tagchecked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
elsif rn_unknown then
    address = bits(64) UNKNOWN;
else
    address = X[n, 64];

if rt_unknown then
    data = bits(elsize) UNKNOWN;
else
    data = X[t, elsize];

bit status = '1';
// Check whether the Exclusives monitors are set to include the
// physical memory locations corresponding to virtual address
// range [address, address+dbytes-1].

// If AArch64.ExclusiveMonitorsPass() returns FALSE and the memory address,
// if accessed, would generate a synchronous Data Abort exception, it is
// IMPLEMENTATION DEFINED whether the exception is generated.
// It is a limitation of this model that synchronous Data Aborts are never
// generated in this case, as Mem[] is not called.
// If FEAT_SPE is implemented, it is also IMPLEMENTATION DEFINED whether or not the
// physical address packet is output when permitted and when
// AArch64.ExclusiveMonitorPass() returns FALSE for a Store Exclusive instruction.
// This behavior is not reflected here due to the previously stated limitation.
```

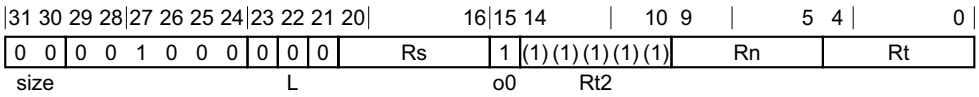
```
if AArch64.ExclusiveMonitorsPass(address, dbytes, accdesc) then
    // This atomic write will be rejected if it does not refer
    // to the same physical locations after address translation.
    Mem[address, dbytes, accdesc] = data;
    status = ExclusiveMonitorsStatus();
X[s, 32] = ZeroExtend(status, 32);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

C6.2.361 STLXRB

Store-Release Exclusive Register Byte stores a byte from a 32-bit register to memory if the PE has exclusive access to the memory address, and returns a status value of 0 if the store was successful, or of 1 if no store was performed. See *Synchronization and semaphores*. The memory access is atomic. The instruction also has memory ordering semantics as described in *Load-Acquire, Load-AcquirePC, and Store-Release*. For information about memory accesses, see *Load/store addressing modes*.



Encoding

```
STLXRB <Ws>, <Wt>, [<Xn|SP>{,#0}]
```

Decode for this encoding

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer s = UInt(Rs);    // ignored by all loads and store-release

boolean tagchecked = n != 31;

boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;
if s == t then
    Constraint c = ConstrainUnpredictable(Unpredictable_DATAOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN rt_unknown = TRUE;    // store UNKNOWN value
        when Constraint_UNDEF   UNDEFINED;
        when Constraint_NOP     EndOfInstruction();
if s == n && n != 31 then
    Constraint c = ConstrainUnpredictable(Unpredictable_BASEOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN rn_unknown = TRUE;    // address is UNKNOWN
        when Constraint_UNDEF   UNDEFINED;
        when Constraint_NOP     EndOfInstruction();
```

Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*, and particularly *STLXRB*.

Assembler symbols

`<Ws>` Is the 32-bit name of the general-purpose register into which the status result of the store exclusive is written, encoded in the "Rs" field. The value returned is:

- `0` If the operation updates memory.
- `1` If the operation fails to update memory.

`<Wt>` Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

`<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Aborts

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- <Ws> is not updated.

If AArch64.ExclusiveMonitorsPass() returns FALSE and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.

Operation

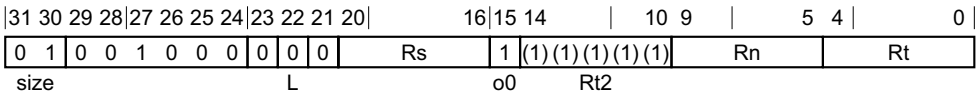
```
bits(64) address;
bits(8) data;

AccessDescriptor accdesc = CreateAccDescExLDST(MemOp_STORE, TRUE, tagchecked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
elsif rn_unknown then
    address = bits(64) UNKNOWN;
else
    address = X[n, 64];

if rt_unknown then
    data = bits(8) UNKNOWN;
else
    data = X[t, 8];

bit status = '1';
// Check whether the Exclusives monitors are set to include the
// physical memory locations corresponding to virtual address
// range [address, address+dbytes-1].

// If AArch64.ExclusiveMonitorsPass() returns FALSE and the memory address,
// if accessed, would generate a synchronous Data Abort exception, it is
// IMPLEMENTATION DEFINED whether the exception is generated.
// It is a limitation of this model that synchronous Data Aborts are never
// generated in this case, as Mem[] is not called.
// If FEAT_SPE is implemented, it is also IMPLEMENTATION DEFINED whether or not the
// physical address packet is output when permitted and when
// AArch64.ExclusiveMonitorPass() returns FALSE for a Store Exclusive instruction.
// This behavior is not reflected here due to the previously stated limitation.
if AArch64.ExclusiveMonitorsPass(address, 1, accdesc) then
    // This atomic write will be rejected if it does not refer
    // to the same physical locations after address translation.
    Mem[address, 1, accdesc] = data;
    status = ExclusiveMonitorsStatus();
X[s, 32] = ZeroExtend(status, 32);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

C6.2.362 STLXRH

Store-Release Exclusive Register Halfword stores a halfword from a 32-bit register to memory if the PE has exclusive access to the memory address, and returns a status value of 0 if the store was successful, or of 1 if no store was performed. See *Synchronization and semaphores*. The memory access is atomic. The instruction also has memory ordering semantics as described in *Load-Acquire, Load-AcquirePC, and Store-Release*. For information about memory accesses, see *Load/store addressing modes*.



Encoding

```
STLXRH <Ws>, <Wt>, [<Xn|SP>{,#0}]
```

Decode for this encoding

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer s = UInt(Rs);    // ignored by all loads and store-release

boolean tagchecked = n != 31;

boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;
if s == t then
    Constraint c = ConstrainUnpredictable(Unpredictable_DATAOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN rt_unknown = TRUE;    // store UNKNOWN value
        when Constraint_UNDEF   UNDEFINED;
        when Constraint_NOP     EndOfInstruction();
if s == n && n != 31 then
    Constraint c = ConstrainUnpredictable(Unpredictable_BASEOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN rn_unknown = TRUE;    // address is UNKNOWN
        when Constraint_UNDEF   UNDEFINED;
        when Constraint_NOP     EndOfInstruction();
```

Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*, and particularly *STLXRH*.

Assembler symbols

`<Ws>` Is the 32-bit name of the general-purpose register into which the status result of the store exclusive is written, encoded in the "Rs" field. The value returned is:

- `0` If the operation updates memory.
- `1` If the operation fails to update memory.

`<Wt>` Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

`<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Aborts and alignment

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- <Ws> is not updated.

A non halfword-aligned memory address causes an Alignment fault Data Abort exception to be generated, subject to the following rules:

- If AArch64.ExclusiveMonitorsPass() returns TRUE, the exception is generated.
- Otherwise, it is IMPLEMENTATION DEFINED whether the exception is generated.

If AArch64.ExclusiveMonitorsPass() returns FALSE and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.

Operation

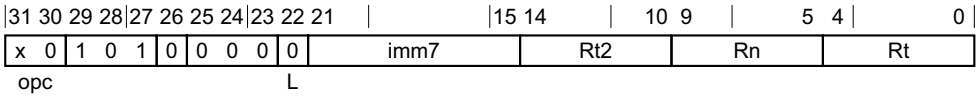
```
bits(64) address;
bits(16) data;

AccessDescriptor accdesc = CreateAccDescExLDST(MemOp_STORE, TRUE, tagchecked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
elsif rn_unknown then
    address = bits(64) UNKNOWN;
else
    address = X[n, 64];

if rt_unknown then
    data = bits(16) UNKNOWN;
else
    data = X[t, 16];

bit status = '1';
// Check whether the Exclusives monitors are set to include the
// physical memory locations corresponding to virtual address
// range [address, address+dbytes-1].

// If AArch64.ExclusiveMonitorsPass() returns FALSE and the memory address,
// if accessed, would generate a synchronous Data Abort exception, it is
// IMPLEMENTATION DEFINED whether the exception is generated.
// It is a limitation of this model that synchronous Data Aborts are never
// generated in this case, as Mem[] is not called.
// If FEAT_SPE is implemented, it is also IMPLEMENTATION DEFINED whether or not the
// physical address packet is output when permitted and when
// AArch64.ExclusiveMonitorPass() returns FALSE for a Store Exclusive instruction.
// This behavior is not reflected here due to the previously stated limitation.
if AArch64.ExclusiveMonitorsPass(address, 2, accdesc) then
    // This atomic write will be rejected if it does not refer
    // to the same physical locations after address translation.
    Mem[address, 2, accdesc] = data;
    status = ExclusiveMonitorsStatus();
X[s, 32] = ZeroExtend(status, 32);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

C6.2.363 STNP

Store Pair of Registers, with non-temporal hint, calculates an address from a base register value and an immediate offset, and stores two 32-bit words or two 64-bit doublewords to the calculated address, from two registers. For information about memory accesses, see *Load/store addressing modes*. For information about Non-temporal pair instructions, see *Load/store non-temporal pair*.



32-bit variant

Applies when `opc == 00`.

```
STNP <Wt1>, <Wt2>, [<Xn|SP>{, #<imm>}]
```

64-bit variant

Applies when `opc == 10`.

```
STNP <Xt1>, <Xt2>, [<Xn|SP>{, #<imm>}]
```

Decode for all variants of this encoding

```
// Empty.
```

Assembler symbols

`<Wt1>` Is the 32-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.

`<Wt2>` Is the 32-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.

`<Xt1>` Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.

`<Xt2>` Is the 64-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.

`<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

`<imm>` For the 32-bit variant: is the optional signed immediate byte offset, a multiple of 4 in the range -256 to 252, defaulting to 0 and encoded in the "imm7" field as <imm>/4.

For the 64-bit variant: is the optional signed immediate byte offset, a multiple of 8 in the range -512 to 504, defaulting to 0 and encoded in the "imm7" field as <imm>/8.

Shared decode for all encodings

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2);
if opc<0> == '1' then UNDEFINED;
integer scale = 2 + UInt(opc<1>);
constant integer datasize = 8 << scale;
bits(64) offset = LSL(SignExtend(imm7, 64), scale);
boolean tagchecked = n != 31;
```

Operation

```
bits(64) address;
bits(64) address2;
bits(datasize) data1;
bits(datasize) data2;
constant integer dbytes = datasize DIV 8;
boolean privileged = PSTATE.EL != EL0;

AccessDescriptor accdesc = CreateAccDescGPR(MemOp_STORE, TRUE, privileged, tagchecked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

address = GenerateAddress(address, offset, accdesc);

data1 = X[t, datasize];
data2 = X[t2, datasize];
address2 = GenerateAddress(address, dbytes, accdesc);
Mem[address, dbytes, accdesc] = data1;
Mem[address2, dbytes, accdesc] = data2;
```

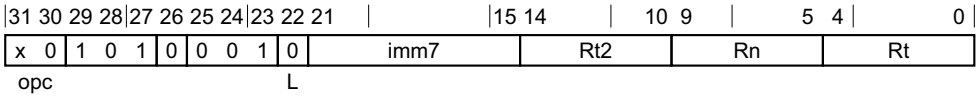
Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

C6.2.364 STP

Store Pair of Registers calculates an address from a base register value and an immediate offset, and stores two 32-bit words or two 64-bit doublewords to the calculated address, from two registers. For information about memory accesses, see *Load/store addressing modes*.

Post-index



32-bit variant

Applies when `opc == 00`.

```
STP <Wt1>, <Wt2>, [<Xn|SP>], #<imm>
```

64-bit variant

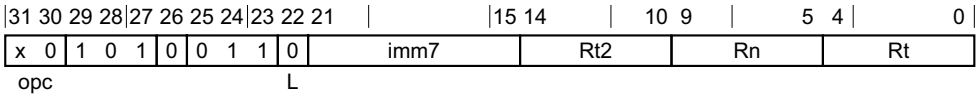
Applies when `opc == 10`.

```
STP <Xt1>, <Xt2>, [<Xn|SP>], #<imm>
```

Decode for all variants of this encoding

```
boolean wback = TRUE;
boolean postindex = TRUE;
```

Pre-index



32-bit variant

Applies when `opc == 00`.

```
STP <Wt1>, <Wt2>, [<Xn|SP>, #<imm>]!
```

64-bit variant

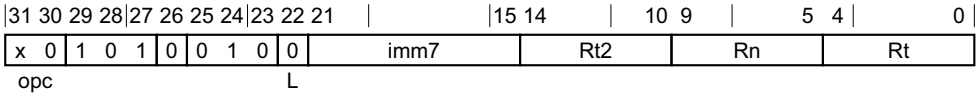
Applies when `opc == 10`.

```
STP <Xt1>, <Xt2>, [<Xn|SP>, #<imm>]!
```

Decode for all variants of this encoding

```
boolean wback = TRUE;
boolean postindex = FALSE;
```

Signed offset



32-bit variant

Applies when opc == 00.

```
STP <Wt1>, <Wt2>, [<Xn|SP>{, #<imm>}]
```

64-bit variant

Applies when opc == 10.

```
STP <Xt1>, <Xt2>, [<Xn|SP>{, #<imm>}]
```

Decode for all variants of this encoding

```
boolean wback = FALSE;
boolean postindex = FALSE;
```

Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*, and particularly *STP and STILP*.

Assembler symbols

<Wt1>	Is the 32-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.
<Wt2>	Is the 32-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.
<Xt1>	Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.
<Xt2>	Is the 64-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm>	For the 32-bit post-index and 32-bit pre-index variant: is the signed immediate byte offset, a multiple of 4 in the range -256 to 252, encoded in the "imm7" field as <imm>/4. For the 32-bit signed offset variant: is the optional signed immediate byte offset, a multiple of 4 in the range -256 to 252, defaulting to 0 and encoded in the "imm7" field as <imm>/4. For the 64-bit post-index and 64-bit pre-index variant: is the signed immediate byte offset, a multiple of 8 in the range -512 to 504, encoded in the "imm7" field as <imm>/8. For the 64-bit signed offset variant: is the optional signed immediate byte offset, a multiple of 8 in the range -512 to 504, defaulting to 0 and encoded in the "imm7" field as <imm>/8.

Shared decode for all encodings

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2);
if L:opc<0> == '01' || opc == '11' then UNDEFINED;
integer scale = 2 + UInt(opc<1>);
constant integer datasize = 8 << scale;
bits(64) offset = LSL(SignExtend(imm7, 64), scale);
boolean tagchecked = wback || n != 31;

boolean rt_unknown = FALSE;

if wback && (t == n || t2 == n) && n != 31 then
    Constraint c = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
```

```
        when Constraint_NONE    rt_unknown = FALSE;    // value stored is pre-writeback
        when Constraint_UNKNOWN rt_unknown = TRUE;    // value stored is UNKNOWN
        when Constraint_UNDEF   UNDEFINED;
        when Constraint_NOP     EndOfInstruction();
```

Operation for all encodings

```
bits(64) address;
bits(64) address2;
bits(datasize) data1;
bits(datasize) data2;
constant integer dbytes = datasize DIV 8;
boolean privileged = PSTATE.EL != EL0;

AccessDescriptor accdesc = CreateAccDescGPR(MemOp_STORE, FALSE, privileged, tagchecked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

if !postindex then
    address = GenerateAddress(address, offset, accdesc);

if rt_unknown && t == n then
    data1 = bits(datasize) UNKNOWN;
else
    data1 = X[t, datasize];
if rt_unknown && t2 == n then
    data2 = bits(datasize) UNKNOWN;
else
    data2 = X[t2, datasize];
if IsFeatureImplemented(FEAT_LSE2) then
    bits(2*datasize) full_data;
    if BigEndian(accdesc.acctype) then
        full_data = data1:data2;
    else
        full_data = data2:data1;
    accdesc.ispair = TRUE;
    Mem[address, 2*dbytes, accdesc] = full_data;
else
    address2 = GenerateAddress(address, dbytes, accdesc);
    Mem[address, dbytes, accdesc] = data1;
    Mem[address2, dbytes, accdesc] = data2;

if wback then
    if postindex then
        address = GenerateAddress(address, offset, accdesc);
    if n == 31 then
        SP[] = address;
    else
        X[n, 64] = address;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

C6.2.365 STR (immediate)

Store Register (immediate) stores a word or a doubleword from a register to memory. The address that is used for the store is calculated from a base register and an immediate offset. For information about memory accesses, see *Load/store addressing modes*.

Post-index

32-bit variant

Applies when `size == 10`.

```
STR <Wt>, [<Xn|SP>], #<simm>
```

64-bit variant

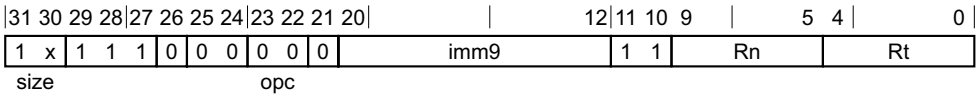
Applies when `size == 11`.

```
STR <Xt>, [<Xn|SP>], #<simm>
```

Decode for all variants of this encoding

```
boolean wback = TRUE;
boolean postindex = TRUE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

Pre-index



32-bit variant

Applies when `size == 10`.

```
STR <Wt>, [<Xn|SP>, #<simm>]!
```

64-bit variant

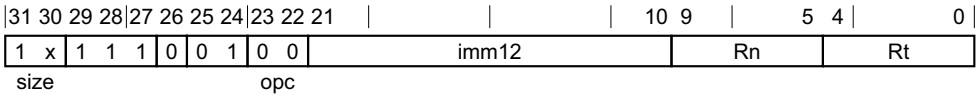
Applies when `size == 11`.

```
STR <Xt>, [<Xn|SP>, #<simm>]!
```

Decode for all variants of this encoding

```
boolean wback = TRUE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

Unsigned offset



32-bit variant

Applies when `size == 10`.

```
STR <Wt>, [<Xn|SP>{, #<pimm>}]
```

64-bit variant

Applies when `size == 11`.

```
STR <Xt>, [<Xn|SP>{, #<pimm>}]
```

Decode for all variants of this encoding

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = LSL(ZeroExtend(imm12, 64), scale);
```

Assembler symbols

`<Wt>`	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
`<Xt>`	Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
`<Xn|SP>`	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
`<simm>`	Is the signed immediate byte offset, in the range -256 to 255, encoded in the "imm9" field.
`<pimm>`	For the 32-bit variant: is the optional positive immediate byte offset, a multiple of 4 in the range 0 to 16380, defaulting to 0 and encoded in the "imm12" field as <pimm>/4. For the 64-bit variant: is the optional positive immediate byte offset, a multiple of 8 in the range 0 to 32760, defaulting to 0 and encoded in the "imm12" field as <pimm>/8.

Shared decode for all encodings

```
integer n = UInt(Rn);
integer t = UInt(Rt);

constant integer datasize = 8 << scale;
boolean tagchecked = wback || n != 31;

boolean rt_unknown = FALSE;
Constraint c;

if wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE    rt_unknown = FALSE;    // value stored is original value
        when Constraint_UNKNOWN rt_unknown = TRUE;    // value stored is UNKNOWN
        when Constraint_UNDEF   UNDEFINED;
        when Constraint_NOP     EndOfInstruction();
```

Operation for all encodings

```
bits(64) address;
bits(datasize) data;

boolean privileged = PSTATE.EL != EL0;
AccessDescriptor accdesc = CreateAccDescGPR(MemOp_STORE, FALSE, privileged, tagchecked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

if !postindex then
    address = GenerateAddress(address, offset, accdesc);

if rt_unknown then
    data = bits(datasize) UNKNOWN;
else
    data = X[t, datasize];
Mem[address, datasize DIV 8, accdesc] = data;

if wback then
    if postindex then
        address = GenerateAddress(address, offset, accdesc);
    if n == 31 then
        SP[] = address;
    else
        X[n, 64] = address;
```

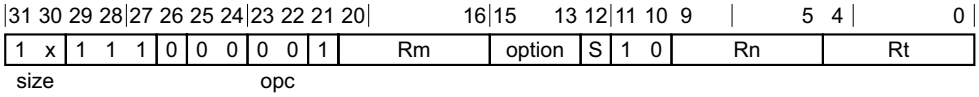
Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

C6.2.366 STR (register)

Store Register (register) calculates an address from a base register value and an offset register value, and stores a 32-bit word or a 64-bit doubleword to the calculated address, from a register. For information about memory accesses, see *Load/store addressing modes*.

The instruction uses an offset addressing mode, that calculates the address used for the memory access from a base register value and an offset register value. The offset can be optionally shifted and extended.



32-bit variant

Applies when `size == 10`.

```
STR <Wt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

64-bit variant

Applies when `size == 11`.

```
STR <Xt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

Decode for all variants of this encoding

```
integer scale = UInt(size);
if option<1> == '0' then UNDEFINED;    // sub-word index
ExtendType extend_type = DecodeRegExtend(option);
integer shift = if S == '1' then scale else 0;
```

Assembler symbols

`<Wt>` Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

`<Xt>` Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

`<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

`<Wm>` When `option<0>` is set to `0`, is the 32-bit name of the general-purpose index register, encoded in the "Rm" field.

`<Xm>` When `option<0>` is set to `1`, is the 64-bit name of the general-purpose index register, encoded in the "Rm" field.

`<extend>` Is the index extend/shift specifier, defaulting to LSL, and which must be omitted for the LSL option when <amount> is omitted. encoded in the "option" field. It can have the following values:

- `UXTW` when `option = 010`
- `LSL` when `option = 011`
- `SXTW` when `option = 110`
- `SXTX` when `option = 111`

`<amount>` For the 32-bit variant: is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in the "S" field. It can have the following values:

- `#0` when `S = 0`

`#2` when `S = 1`

For the 64-bit variant: is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in the "S" field. It can have the following values:

`#0` when `S = 0`

`#3` when `S = 1`

Shared decode for all encodings

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer m = UInt(Rm);

constant integer datasize = 8 << scale;
```

Operation

```
bits(64) offset = ExtendReg(m, extend_type, shift, 64);
bits(64) address;
bits(datasize) data;

boolean privileged = PSTATE.EL != EL0;
AccessDescriptor accdesc = CreateAccDescGPR(MemOp_STORE, FALSE, privileged, TRUE);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

address = GenerateAddress(address, offset, accdesc);

data = X[t, datasize];
Mem[address, datasize DIV 8, accdesc] = data;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

C6.2.367 STRB (immediate)

Store Register Byte (immediate) stores the least significant byte of a 32-bit register to memory. The address that is used for the store is calculated from a base register and an immediate offset. For information about memory accesses, see *Load/store addressing modes*.

Post-index

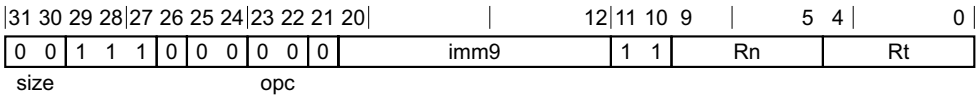
Encoding

```
STRB <Wt>, [<Xn|SP>], #<simm>
```

Decode for this encoding

```
boolean wback = TRUE;
boolean postindex = TRUE;
bits(64) offset = SignExtend(imm9, 64);
```

Pre-index



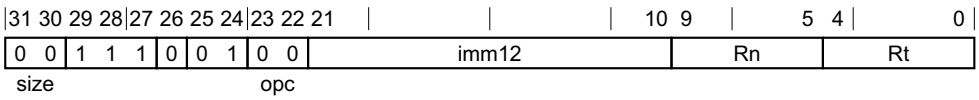
Encoding

```
STRB <Wt>, [<Xn|SP>, #<simm>]!
```

Decode for this encoding

```
boolean wback = TRUE;
boolean postindex = FALSE;
bits(64) offset = SignExtend(imm9, 64);
```

Unsigned offset



Encoding

```
STRB <Wt>, [<Xn|SP>{, #<pimm>}]
```

Decode for this encoding

```
boolean wback = FALSE;
boolean postindex = FALSE;
bits(64) offset = LSL(ZeroExtend(imm12, 64), 0);
```

Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*, and particularly *STRB (immediate)*.

Assembler symbols

<Wt>	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<simm>	Is the signed immediate byte offset, in the range -256 to 255, encoded in the "imm9" field.
<pimm>	Is the optional positive immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the "imm12" field.

Shared decode for all encodings

```
integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tagchecked = wback || n != 31;

boolean rt_unknown = FALSE;
Constraint c;

if wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE    rt_unknown = FALSE;    // value stored is original value
        when Constraint_UNKNOWN rt_unknown = TRUE;    // value stored is UNKNOWN
        when Constraint_UNDEF   UNDEFINED;
        when Constraint_NOP     EndOfInstruction();
```

Operation for all encodings

```
bits(64) address;
bits(8) data;

boolean privileged = PSTATE.EL != EL0;
AccessDescriptor accdesc = CreateAccDescGPR(MemOp_STORE, FALSE, privileged, tagchecked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

if !postindex then
    address = GenerateAddress(address, offset, accdesc);

if rt_unknown then
    data = bits(8) UNKNOWN;
else
    data = X[t, 8];
Mem[address, 1, accdesc] = data;

if wback then
    if postindex then
        address = GenerateAddress(address, offset, accdesc);
    if n == 31 then
        SP[] = address;
```

```
else
    X[n, 64] = address;
```

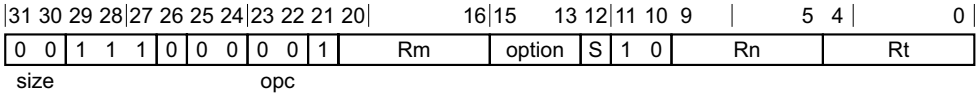
Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

C6.2.368 STRB (register)

Store Register Byte (register) calculates an address from a base register value and an offset register value, and stores a byte from a 32-bit register to the calculated address. For information about memory accesses, see *Load/store addressing modes*.

The instruction uses an offset addressing mode, that calculates the address used for the memory access from a base register value and an offset register value. The offset can be optionally shifted and extended.



Extended register variant

Applies when `option != 011`.

```
STRB <Wt>, [<Xn|SP>, (<Wm>|<Xm>), <extend> {<amount>}]
```

Shifted register variant

Applies when `option == 011`.

```
STRB <Wt>, [<Xn|SP>, <Xm>{, LSL <amount>}]
```

Decode for all variants of this encoding

```
if option<1> == '0' then UNDEFINED;    // sub-word index
ExtendType extend_type = DecodeRegExtend(option);
```

Assembler symbols

`<Wt>` Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

`<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

`<Wm>` When `option<0>` is set to `0`, is the 32-bit name of the general-purpose index register, encoded in the "Rm" field.

`<Xm>` When `option<0>` is set to `1`, is the 64-bit name of the general-purpose index register, encoded in the "Rm" field.

`<extend>` Is the index extend specifier, encoded in the "option" field. It can have the following values:

- `UXTW` when `option = 010`
- `SXTW` when `option = 110`
- `SXTX` when `option = 111`

`<amount>` Is the index shift amount, it must be `#0`, encoded in "S" as `0` if omitted, or as `1` if present.

Shared decode for all encodings

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer m = UInt(Rm);
```

Operation

```
bits(64) offset = ExtendReg(m, extend_type, 0, 64);
bits(64) address;
bits(8) data;

boolean privileged = PSTATE.EL != EL0;
AccessDescriptor accdesc = CreateAccDescGPR(MemOp_STORE, FALSE, privileged, TRUE);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

address = GenerateAddress(address, offset, accdesc);

data = X[t, 8];
Mem[address, 1, accdesc] = data;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

C6.2.369 STRH (immediate)

Store Register Halfword (immediate) stores the least significant halfword of a 32-bit register to memory. The address that is used for the store is calculated from a base register and an immediate offset. For information about memory accesses, see *Load/store addressing modes*.

Post-index

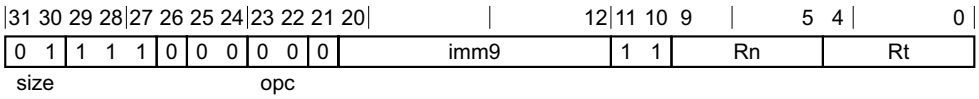
Encoding

```
STRH <Wt>, [<Xn|SP>], #<simm>
```

Decode for this encoding

```
boolean wback = TRUE;
boolean postindex = TRUE;
bits(64) offset = SignExtend(imm9, 64);
```

Pre-index



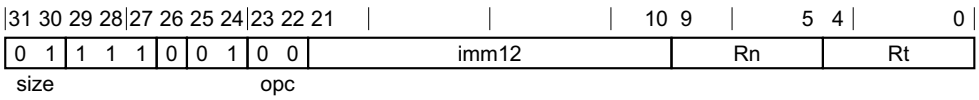
Encoding

```
STRH <Wt>, [<Xn|SP>, #<simm>]!
```

Decode for this encoding

```
boolean wback = TRUE;
boolean postindex = FALSE;
bits(64) offset = SignExtend(imm9, 64);
```

Unsigned offset



Encoding

```
STRH <Wt>, [<Xn|SP>{, #<pimm>}]
```

Decode for this encoding

```
boolean wback = FALSE;
boolean postindex = FALSE;
bits(64) offset = LSL(ZeroExtend(imm12, 64), 1);
```

Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*, and particularly *STRH (immediate)*.

Assembler symbols

<Wt>	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<simm>	Is the signed immediate byte offset, in the range -256 to 255, encoded in the "imm9" field.
<pimm>	Is the optional positive immediate byte offset, a multiple of 2 in the range 0 to 8190, defaulting to 0 and encoded in the "imm12" field as <pimm>/2.

Shared decode for all encodings

```
integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tagchecked = wback || n != 31;

boolean rt_unknown = FALSE;
Constraint c;

if wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE    rt_unknown = FALSE;    // value stored is original value
        when Constraint_UNKNOWN rt_unknown = TRUE;    // value stored is UNKNOWN
        when Constraint_UNDEF   UNDEFINED;
        when Constraint_NOP     EndOfInstruction();
```

Operation for all encodings

```
bits(64) address;
bits(16) data;

boolean privileged = PSTATE.EL != EL0;
AccessDescriptor accdesc = CreateAccDescGPR(MemOp_STORE, FALSE, privileged, tagchecked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

if !postindex then
    address = GenerateAddress(address, offset, accdesc);

if rt_unknown then
    data = bits(16) UNKNOWN;
else
    data = X[t, 16];
Mem[address, 2, accdesc] = data;

if wback then
    if postindex then
        address = GenerateAddress(address, offset, accdesc);
    if n == 31 then
        SP[] = address;
```

```
else
    X[n, 64] = address;
```

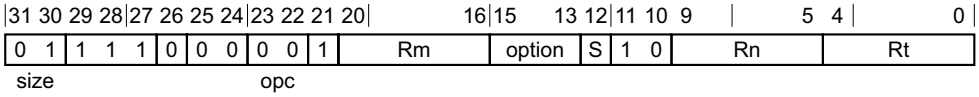
Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

C6.2.370 STRH (register)

Store Register Halfword (register) calculates an address from a base register value and an offset register value, and stores a halfword from a 32-bit register to the calculated address. For information about memory accesses, see *Load/store addressing modes*.

The instruction uses an offset addressing mode, that calculates the address used for the memory access from a base register value and an offset register value. The offset can be optionally shifted and extended.



Encoding

```
STRH <Wt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

Decode for this encoding

```
if option<1> == '0' then UNDEFINED;    // sub-word index
ExtendType extend_type = DecodeRegExtend(option);
integer shift = if S == '1' then 1 else 0;
```

Assembler symbols

`<Wt>` Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

`<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

`<Wm>` When `option<0>` is set to `0`, is the 32-bit name of the general-purpose index register, encoded in the "Rm" field.

`<Xm>` When `option<0>` is set to `1`, is the 64-bit name of the general-purpose index register, encoded in the "Rm" field.

`<extend>` Is the index extend/shift specifier, defaulting to LSL, and which must be omitted for the LSL option when <amount> is omitted. encoded in the "option" field. It can have the following values:

- `UXTW` when `option = 010`
- `LSL` when `option = 011`
- `SXTW` when `option = 110`
- `SXTX` when `option = 111`

`<amount>` Is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in the "S" field. It can have the following values:

- `#0` when `S = 0`
- `#1` when `S = 1`

Shared decode for all encodings

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer m = UInt(Rm);
```

Operation

```
bits(64) offset = ExtendReg(m, extend_type, shift, 64);
bits(64) address;
bits(16) data;

boolean privileged = PSTATE.EL != EL0;
AccessDescriptor accdesc = CreateAccDescGPR(MemOp_STORE, FALSE, privileged, TRUE);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

address = GenerateAddress(address, offset, accdesc);

data = X[t, 16];
Mem[address, 2, accdesc] = data;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

C6.2.371 STSET, STSETL

Atomic bit set on word or doubleword in memory, without return, atomically loads a 32-bit word or 64-bit doubleword from memory, performs a bitwise OR with the value held in a register on it, and stores the result back to memory.

- STSET does not have release semantics.
- STSETL stores to memory with release semantics, as described in *Load-Acquire, Load-AcquirePC, and Store-Release*.

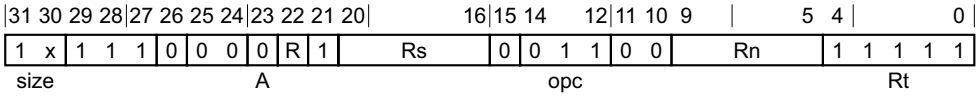
For information about memory accesses, see *Load/store addressing modes*.

This instruction is an alias of the LDSET, LDSETA, LDSETAL, LDSETL instruction. This means that:

- The encodings in this description are named to match the encodings of LDSET, LDSETA, LDSETAL, LDSETL.
- The description of LDSET, LDSETA, LDSETAL, LDSETL gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Integer

(FEAT_LSE)



32-bit LDSET alias variant

Applies when `size == 10 && R == 0`.

```
STSET <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDSET <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

32-bit LDSETL alias variant

Applies when `size == 10 && R == 1`.

```
STSETL <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDSETL <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

64-bit LDSET alias variant

Applies when `size == 11 && R == 0`.

```
STSET <Xs>, [<Xn|SP>]
```

is equivalent to

```
LDSET <Xs>, XZR, [<Xn|SP>]
```

and is always the preferred disassembly.

64-bit LDSETL alias variant

Applies when `size == 11 && R == 1`.

```
STSETL <Xs>, [<Xn|SP>]
```

is equivalent to

```
LDSETL <Xs>, XZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Assembler symbols

`<Ws>`	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
`<Xs>`	Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
`<Xn|SP>`	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

The description of LDSET, LDSETA, LDSETAL, LDSETL gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

C6.2.372 STSETB, STSETLB

Atomic bit set on byte in memory, without return, atomically loads an 8-bit byte from memory, performs a bitwise OR with the value held in a register on it, and stores the result back to memory.

- STSETB does not have release semantics.
- STSETLB stores to memory with release semantics, as described in *Load-Acquire, Load-AcquirePC, and Store-Release*.

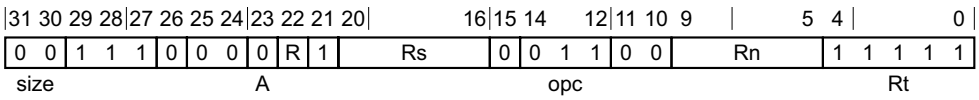
For information about memory accesses, see *Load/store addressing modes*.

This instruction is an alias of the LDSETB, LDSETAB, LDSETALB, LDSETLB instruction. This means that:

- The encodings in this description are named to match the encodings of LDSETB, LDSETAB, LDSETALB, LDSETLB.
- The description of LDSETB, LDSETAB, LDSETALB, LDSETLB gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Integer

(FEAT_LSE)



No memory ordering variant

Applies when R == 0.

```
STSETB <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDSETB <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Release variant

Applies when R == 1.

```
STSETLB <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDSETLB <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Assembler symbols

<Ws>	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

The description of LDSETB, LDSETAB, LDSETALB, LDSETLB gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

C6.2.373 STSETH, STSETLH

Atomic bit set on halfword in memory, without return, atomically loads a 16-bit halfword from memory, performs a bitwise OR with the value held in a register on it, and stores the result back to memory.

- STSETH does not have release semantics.
- STSETLH stores to memory with release semantics, as described in *Load-Acquire, Load-AcquirePC, and Store-Release*.

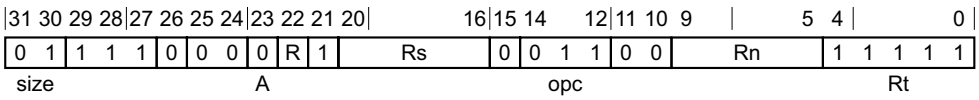
For information about memory accesses, see *Load/store addressing modes*.

This instruction is an alias of the LDSETH, LDSETAH, LDSETALH, LDSETLH instruction. This means that:

- The encodings in this description are named to match the encodings of LDSETH, LDSETAH, LDSETALH, LDSETLH.
- The description of LDSETH, LDSETAH, LDSETALH, LDSETLH gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Integer

(FEAT_LSE)



No memory ordering variant

Applies when R == 0.

```
STSETH <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDSETH <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Release variant

Applies when R == 1.

```
STSETLH <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDSETLH <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Assembler symbols

<Ws>	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

The description of LDSETH, LDSETAH, LDSETALH, LDSETLH gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

C6.2.374 STSMAX, STSMAXL

Atomic signed maximum on word or doubleword in memory, without return, atomically loads a 32-bit word or 64-bit doubleword from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as signed numbers.

- `STSMAX` does not have release semantics.
- `STSMAXL` stores to memory with release semantics, as described in *Load-Acquire, Load-AcquirePC, and Store-Release*.

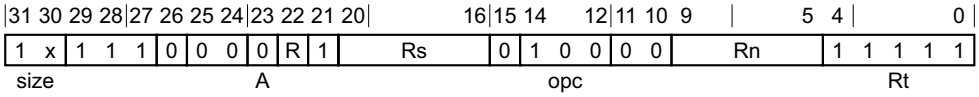
For information about memory accesses, see *Load/store addressing modes*.

This instruction is an alias of the LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL instruction. This means that:

- The encodings in this description are named to match the encodings of LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL.
- The description of LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Integer

(FEAT_LSE)



32-bit LDSMAX alias variant

Applies when `size == 10 && R == 0`.

```
STSMAX <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDSMAX <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

32-bit LDSMAXL alias variant

Applies when `size == 10 && R == 1`.

```
STSMAXL <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDSMAXL <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

64-bit LDSMAX alias variant

Applies when `size == 11 && R == 0`.

```
STSMAX <Xs>, [<Xn|SP>]
```

is equivalent to

```
LDSMAX <Xs>, XZR, [<Xn|SP>]
```

and is always the preferred disassembly.

64-bit LDSMAXL alias variant

Applies when `size == 11 && R == 1`.

```
STSMAXL <Xs>, [<Xn|SP>]
```

is equivalent to

```
LDSMAXL <Xs>, XZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Assembler symbols

`<Ws>`	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
`<Xs>`	Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
`<Xn|SP>`	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

The description of LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

C6.2.375 STSMAXB, STSMAXLB

Atomic signed maximum on byte in memory, without return, atomically loads an 8-bit byte from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as signed numbers.

- `STSMAXB` does not have release semantics.
- `STSMAXLB` stores to memory with release semantics, as described in *Load-Acquire, Load-AcquirePC, and Store-Release*.

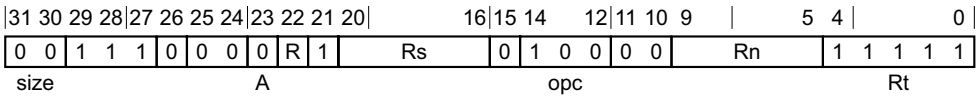
For information about memory accesses, see *Load/store addressing modes*.

This instruction is an alias of the LDSMAXB, LDSMAXAB, LDSMAXALB, LDSMAXLB instruction. This means that:

- The encodings in this description are named to match the encodings of LDSMAXB, LDSMAXAB, LDSMAXALB, LDSMAXLB.
- The description of LDSMAXB, LDSMAXAB, LDSMAXALB, LDSMAXLB gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Integer

(FEAT_LSE)



No memory ordering variant

Applies when `R == 0`.

```
STSMAXB <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDSMAXB <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Release variant

Applies when `R == 1`.

```
STSMAXLB <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDSMAXLB <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Assembler symbols

`<Ws>` Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

`<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

The description of LDSMAXB, LDSMAXAB, LDSMAXALB, LDSMAXLB gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

C6.2.376 STSMAXH, STSMAXLH

Atomic signed maximum on halfword in memory, without return, atomically loads a 16-bit halfword from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as signed numbers.

- `STSMAXH` does not have release semantics.
- `STSMAXLH` stores to memory with release semantics, as described in *Load-Acquire, Load-AcquirePC, and Store-Release*.

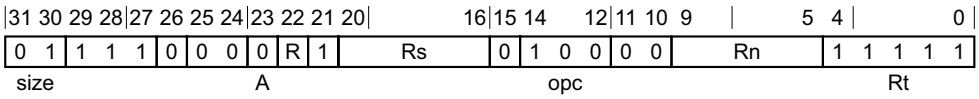
For information about memory accesses, see *Load/store addressing modes*.

This instruction is an alias of the LDSMAXH, LDSMAXAH, LDSMAXALH, LDSMAXLH instruction. This means that:

- The encodings in this description are named to match the encodings of LDSMAXH, LDSMAXAH, LDSMAXALH, LDSMAXLH.
- The description of LDSMAXH, LDSMAXAH, LDSMAXALH, LDSMAXLH gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Integer

(FEAT_LSE)



No memory ordering variant

Applies when `R == 0`.

```
STSMAXH <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDSMAXH <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Release variant

Applies when `R == 1`.

```
STSMAXLH <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDSMAXLH <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Assembler symbols

`<Ws>` Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

`<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

The description of LDSMAXH, LDSMAXAH, LDSMAXALH, LDSMAXLH gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

C6.2.377 STSMIN, STSMINL

Atomic signed minimum on word or doubleword in memory, without return, atomically loads a 32-bit word or 64-bit doubleword from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as signed numbers.

- `STSMIN` does not have release semantics.
- `STSMINL` stores to memory with release semantics, as described in *Load-Acquire, Load-AcquirePC, and Store-Release*.

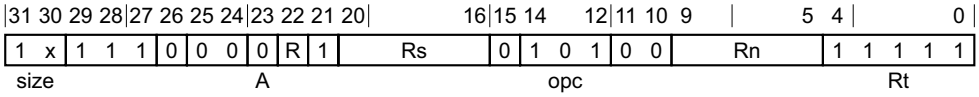
For information about memory accesses, see *Load/store addressing modes*.

This instruction is an alias of the LDSMIN, LDSMINA, LDSMINAL, LDSMINL instruction. This means that:

- The encodings in this description are named to match the encodings of LDSMIN, LDSMINA, LDSMINAL, LDSMINL.
- The description of LDSMIN, LDSMINA, LDSMINAL, LDSMINL gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Integer

(FEAT_LSE)



32-bit LDSMIN alias variant

Applies when `size == 10 && R == 0`.

```
STSMIN <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDSMIN <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

32-bit LDSMINL alias variant

Applies when `size == 10 && R == 1`.

```
STSMINL <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDSMINL <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

64-bit LDSMIN alias variant

Applies when `size == 11 && R == 0`.

```
STSMIN <Xs>, [<Xn|SP>]
```

is equivalent to

```
LDSMIN <Xs>, XZR, [<Xn|SP>]
```

and is always the preferred disassembly.

64-bit LDSMINL alias variant

Applies when `size == 11 && R == 1`.

```
STSMINL <Xs>, [<Xn|SP>]
```

is equivalent to

```
LDSMINL <Xs>, XZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Assembler symbols

`<Ws>`	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
`<Xs>`	Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
`<Xn|SP>`	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

The description of LDSMIN, LDSMINA, LDSMINAL, LDSMINL gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

C6.2.378 STSMINB, STSMINLB

Atomic signed minimum on byte in memory, without return, atomically loads an 8-bit byte from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as signed numbers.

- `STSMINB` does not have release semantics.
- `STSMINLB` stores to memory with release semantics, as described in *Load-Acquire, Load-AcquirePC, and Store-Release*.

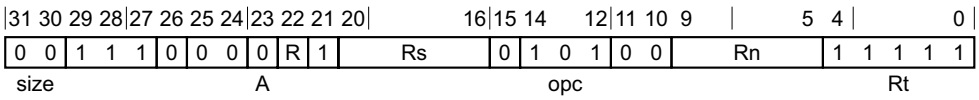
For information about memory accesses, see *Load/store addressing modes*.

This instruction is an alias of the LDSMINB, LDSMINAB, LDSMINALB, LDSMINLB instruction. This means that:

- The encodings in this description are named to match the encodings of LDSMINB, LDSMINAB, LDSMINALB, LDSMINLB.
- The description of LDSMINB, LDSMINAB, LDSMINALB, LDSMINLB gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Integer

(FEAT_LSE)



No memory ordering variant

Applies when `R == 0`.

```
STSMINB <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDSMINB <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Release variant

Applies when `R == 1`.

```
STSMINLB <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDSMINLB <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Assembler symbols

`<Ws>`	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
`<Xn|SP>`	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

The description of LDSMINB, LDSMINAB, LDSMINALB, LDSMINLB gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

C6.2.379 STSMINH, STSMINLH

Atomic signed minimum on halfword in memory, without return, atomically loads a 16-bit halfword from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as signed numbers.

- `STSMINH` does not have release semantics.
- `STSMINLH` stores to memory with release semantics, as described in *Load-Acquire, Load-AcquirePC, and Store-Release*.

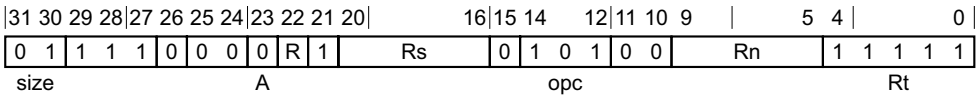
For information about memory accesses, see *Load/store addressing modes*.

This instruction is an alias of the LDSMINH, LDSMINAH, LDSMINALH, LDSMINLH instruction. This means that:

- The encodings in this description are named to match the encodings of LDSMINH, LDSMINAH, LDSMINALH, LDSMINLH.
- The description of LDSMINH, LDSMINAH, LDSMINALH, LDSMINLH gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Integer

(FEAT_LSE)



No memory ordering variant

Applies when `R == 0`.

```
STSMINH <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDSMINH <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Release variant

Applies when `R == 1`.

```
STSMINLH <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDSMINLH <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Assembler symbols

`<Ws>`	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
`<Xn|SP>`	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

The description of LDSMINH, LDSMINAH, LDSMINALH, LDSMINLH gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

C6.2.380 STTR

Store Register (unprivileged) stores a word or doubleword from a register to memory. The address that is used for the store is calculated from a base register and an immediate offset.

Memory accesses made by the instruction behave as if the instruction was executed at EL0 if the *Effective value* of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the *Effective value* of HCR_EL2.{E2H, TGE} is {1, 1}.

Otherwise, the memory access operates with the restrictions determined by the Exception level at which the instruction is executed. For information about memory accesses, see *Load/store addressing modes*.

32-bit variant

Applies when `size == 10`.

```
STTR <Wt>, [<Xn|SP>{, #<simm>}]
```

64-bit variant

Applies when `size == 11`.

```
STTR <Xt>, [<Xn|SP>{, #<simm>}]
```

Decode for all variants of this encoding

```
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

Assembler symbols

`<Wt>`	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
`<Xt>`	Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
`<Xn|SP>`	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
`<simm>`	Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared decode for all encodings

```
integer n = UInt(Rn);
integer t = UInt(Rt);

constant integer datasize = 8 << scale;
boolean tagchecked = n != 31;
```

Operation

```
bits(64) address;
bits(datasize) data;

boolean privileged = AArch64.IsUnprivAccessPriv();
AccessDescriptor accdesc = CreateAccDescGPR(MemOp_STORE, FALSE, privileged, tagchecked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

address = GenerateAddress(address, offset, accdesc);

data = X[t, datasize];
Mem[address, datasize DIV 8, accdesc] = data;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

C6.2.381 STTRB

Store Register Byte (unprivileged) stores a byte from a 32-bit register to memory. The address that is used for the store is calculated from a base register and an immediate offset.

Memory accesses made by the instruction behave as if the instruction was executed at EL0 if the *Effective value* of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the *Effective value* of HCR_EL2.{E2H, TGE} is {1, 1}.

Otherwise, the memory access operates with the restrictions determined by the Exception level at which the instruction is executed. For information about memory accesses, see *Load/store addressing modes*.

Encoding

```
STTRB <Wt>, [<Xn|SP>{, #<simm>}]
```

Decode for this encoding

```
bits(64) offset = SignExtend(imm9, 64);
```

Assembler symbols

<Wt>	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<simm>	Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared decode for all encodings

```
integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tagchecked = n != 31;
```

Operation

```
bits(64) address;
bits(8) data;

boolean privileged = AArch64.IsUnprivAccessPriv();
AccessDescriptor accdesc = CreateAccDescGPR(MemOp_STORE, FALSE, privileged, tagchecked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

address = GenerateAddress(address, offset, accdesc);
```

```
data = X[t, 8];
Mem[address, 1, accdesc] = data;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

C6.2.382 STTRH

Store Register Halfword (unprivileged) stores a halfword from a 32-bit register to memory. The address that is used for the store is calculated from a base register and an immediate offset.

Memory accesses made by the instruction behave as if the instruction was executed at EL0 if the *Effective value* of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the *Effective value* of HCR_EL2.{E2H, TGE} is {1, 1}.

Otherwise, the memory access operates with the restrictions determined by the Exception level at which the instruction is executed. For information about memory accesses, see *Load/store addressing modes*.

Encoding

```
STTRH <Wt>, [<Xn|SP>{, #<simm>}]
```

Decode for this encoding

```
bits(64) offset = SignExtend(imm9, 64);
```

Assembler symbols

<Wt>	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<simm>	Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared decode for all encodings

```
integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tagchecked = n != 31;
```

Operation

```
bits(64) address;
bits(16) data;

boolean privileged = AArch64.IsUnprivAccessPriv();
AccessDescriptor accdesc = CreateAccDescGPR(MemOp_STORE, FALSE, privileged, tagchecked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

address = GenerateAddress(address, offset, accdesc);
```

```
data = X[t, 16];
Mem[address, 2, accdesc] = data;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

C6.2.383 STUMAX, STUMAXL

Atomic unsigned maximum on word or doubleword in memory, without return, atomically loads a 32-bit word or 64-bit doubleword from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as unsigned numbers.

- `STUMAX` does not have release semantics.
- `STUMAXL` stores to memory with release semantics, as described in *Load-Acquire, Load-AcquirePC, and Store-Release*.

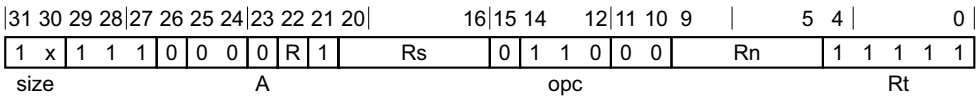
For information about memory accesses, see *Load/store addressing modes*.

This instruction is an alias of the LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL instruction. This means that:

- The encodings in this description are named to match the encodings of LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL.
- The description of LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Integer

(FEAT_LSE)



32-bit LDUMAX alias variant

Applies when `size == 10 && R == 0`.

```
STUMAX <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDUMAX <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

32-bit LDUMAXL alias variant

Applies when `size == 10 && R == 1`.

```
STUMAXL <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDUMAXL <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

64-bit LDUMAX alias variant

Applies when `size == 11 && R == 0`.

```
STUMAX <Xs>, [<Xn|SP>]
```

is equivalent to

```
LDUMAX <Xs>, XZR, [<Xn|SP>]
```

and is always the preferred disassembly.

64-bit LDUMAXL alias variant

Applies when `size == 11 && R == 1`.

```
STUMAXL <Xs>, [<Xn|SP>]
```

is equivalent to

```
LDUMAXL <Xs>, XZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Assembler symbols

`<Ws>`	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
`<Xs>`	Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
`<Xn|SP>`	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

The description of LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

C6.2.384 STUMAXB, STUMAXLB

Atomic unsigned maximum on byte in memory, without return, atomically loads an 8-bit byte from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as unsigned numbers.

- `STUMAXB` does not have release semantics.
- `STUMAXLB` stores to memory with release semantics, as described in *Load-Acquire, Load-AcquirePC, and Store-Release*.

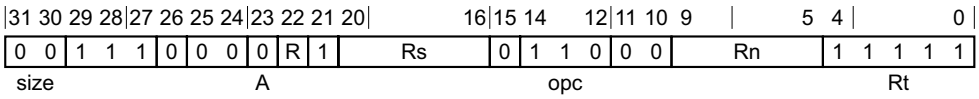
For information about memory accesses, see *Load/store addressing modes*.

This instruction is an alias of the LDUMAXB, LDUMAXAB, LDUMAXALB, LDUMAXLB instruction. This means that:

- The encodings in this description are named to match the encodings of LDUMAXB, LDUMAXAB, LDUMAXALB, LDUMAXLB.
- The description of LDUMAXB, LDUMAXAB, LDUMAXALB, LDUMAXLB gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Integer

(FEAT_LSE)



No memory ordering variant

Applies when `R == 0`.

```
STUMAXB <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDUMAXB <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Release variant

Applies when `R == 1`.

```
STUMAXLB <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDUMAXLB <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Assembler symbols

`<Ws>`	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
`<Xn|SP>`	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

The description of LDUMAXB, LDUMAXAB, LDUMAXALB, LDUMAXLB gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

C6.2.385 STUMAXH, STUMAXLH

Atomic unsigned maximum on halfword in memory, without return, atomically loads a 16-bit halfword from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as unsigned numbers.

- `STUMAXH` does not have release semantics.
- `STUMAXLH` stores to memory with release semantics, as described in *Load-Acquire, Load-AcquirePC, and Store-Release*.

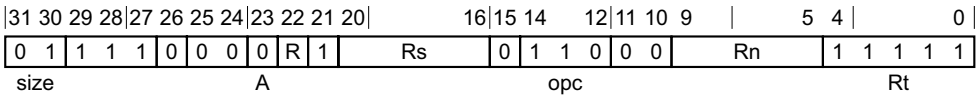
For information about memory accesses see *Load/store addressing modes*.

This instruction is an alias of the LDUMAXH, LDUMAXAH, LDUMAXALH, LDUMAXLH instruction. This means that:

- The encodings in this description are named to match the encodings of LDUMAXH, LDUMAXAH, LDUMAXALH, LDUMAXLH.
- The description of LDUMAXH, LDUMAXAH, LDUMAXALH, LDUMAXLH gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Integer

(FEAT_LSE)



No memory ordering variant

Applies when `R == 0`.

```
STUMAXH <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDUMAXH <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Release variant

Applies when `R == 1`.

```
STUMAXLH <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDUMAXLH <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Assembler symbols

`<Ws>`	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
`<Xn|SP>`	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

The description of LDUMAXH, LDUMAXAH, LDUMAXALH, LDUMAXLH gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

C6.2.386 STUMIN, STUMINL

Atomic unsigned minimum on word or doubleword in memory, without return, atomically loads a 32-bit word or 64-bit doubleword from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as unsigned numbers.

- `STUMIN` does not have release semantics.
- `STUMINL` stores to memory with release semantics, as described in *Load-Acquire, Load-AcquirePC, and Store-Release*.

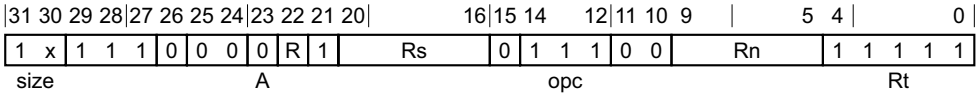
For information about memory accesses, see *Load/store addressing modes*.

This instruction is an alias of the LDUMIN, LDUMINA, LDUMINAL, LDUMINL instruction. This means that:

- The encodings in this description are named to match the encodings of LDUMIN, LDUMINA, LDUMINAL, LDUMINL.
- The description of LDUMIN, LDUMINA, LDUMINAL, LDUMINL gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Integer

(FEAT_LSE)



32-bit LDUMIN alias variant

Applies when `size == 10 && R == 0`.

```
STUMIN <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDUMIN <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

32-bit LDUMINL alias variant

Applies when `size == 10 && R == 1`.

```
STUMINL <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDUMINL <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

64-bit LDUMIN alias variant

Applies when `size == 11 && R == 0`.

```
STUMIN <Xs>, [<Xn|SP>]
```

is equivalent to

```
LDUMIN <Xs>, XZR, [<Xn|SP>]
```

and is always the preferred disassembly.

64-bit LDUMINL alias variant

Applies when `size == 11 && R == 1`.

```
STUMINL <Xs>, [<Xn|SP>]
```

is equivalent to

```
LDUMINL <Xs>, XZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Assembler symbols

`<Ws>`	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
`<Xs>`	Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
`<Xn|SP>`	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

The description of LDUMIN, LDUMINA, LDUMINAL, LDUMINL gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

C6.2.387 STUMINB, STUMINLB

Atomic unsigned minimum on byte in memory, without return, atomically loads an 8-bit byte from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as unsigned numbers.

- `STUMINB` does not have release semantics.
- `STUMINLB` stores to memory with release semantics, as described in *Load-Acquire, Load-AcquirePC, and Store-Release*.

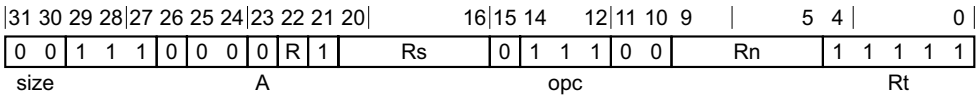
For information about memory accesses, see *Load/store addressing modes*.

This instruction is an alias of the LDUMINB, LDUMINAB, LDUMINALB, LDUMINLB instruction. This means that:

- The encodings in this description are named to match the encodings of LDUMINB, LDUMINAB, LDUMINALB, LDUMINLB.
- The description of LDUMINB, LDUMINAB, LDUMINALB, LDUMINLB gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Integer

(FEAT_LSE)



No memory ordering variant

Applies when `R == 0`.

```
STUMINB <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDUMINB <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Release variant

Applies when `R == 1`.

```
STUMINLB <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDUMINLB <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Assembler symbols

`<Ws>`	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
`<Xn|SP>`	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

The description of LDUMINB, LDUMINAB, LDUMINALB, LDUMINLB gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

C6.2.388 STUMINH, STUMINLH

Atomic unsigned minimum on halfword in memory, without return, atomically loads a 16-bit halfword from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as unsigned numbers.

- `STUMINH` does not have release semantics.
- `STUMINLH` stores to memory with release semantics, as described in *Load-Acquire, Load-AcquirePC, and Store-Release*.

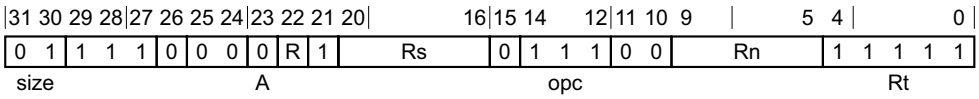
For information about memory accesses, see *Load/store addressing modes*.

This instruction is an alias of the LDUMINH, LDUMINAH, LDUMINALH, LDUMINLH instruction. This means that:

- The encodings in this description are named to match the encodings of LDUMINH, LDUMINAH, LDUMINALH, LDUMINLH.
- The description of LDUMINH, LDUMINAH, LDUMINALH, LDUMINLH gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

Integer

(FEAT_LSE)



No memory ordering variant

Applies when `R == 0`.

```
STUMINH <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDUMINH <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Release variant

Applies when `R == 1`.

```
STUMINLH <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDUMINLH <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Assembler symbols

`<Ws>` Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

`<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

The description of LDUMINH, LDUMINAH, LDUMINALH, LDUMINLH gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

C6.2.389 STUR

Store Register (unscaled) calculates an address from a base register value and an immediate offset, and stores a 32-bit word or a 64-bit doubleword to the calculated address, from a register. For information about memory accesses, see *Load/store addressing modes*.

32-bit variant

Applies when size == 10.

```
STUR <Wt>, [<Xn|SP>{, #<simm>}]
```

64-bit variant

Applies when size == 11.

```
STUR <Xt>, [<Xn|SP>{, #<simm>}]
```

Decode for all variants of this encoding

```
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

Assembler symbols

<Wt>	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xt>	Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<simm>	Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared decode for all encodings

```
integer n = UInt(Rn);
integer t = UInt(Rt);

constant integer datasize = 8 << scale;
boolean tagchecked = n != 31;
```

Operation

```
bits(64) address;
bits(datasize) data;

boolean privileged = PSTATE.EL != EL0;
AccessDescriptor accdesc = CreateAccDescGPR(MemOp_STORE, FALSE, privileged, tagchecked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
```

```
    address = X[n, 64];

address = GenerateAddress(address, offset, accdesc);

data = X[t, datasize];
Mem[address, datasize DIV 8, accdesc] = data;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

C6.2.390 STURB

Store Register Byte (unscaled) calculates an address from a base register value and an immediate offset, and stores a byte to the calculated address, from a 32-bit register. For information about memory accesses, see *Load/store addressing modes*.

Encoding

```
STURB <Wt>, [<Xn|SP>{, #<simm>}]
```

Decode for this encoding

```
bits(64) offset = SignExtend(imm9, 64);
```

Assembler symbols

<Wt>	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<simm>	Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared decode for all encodings

```
integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tagchecked = n != 31;
```

Operation

```
bits(64) address;
bits(8) data;

boolean privileged = PSTATE.EL != EL0;
AccessDescriptor accdesc = CreateAccDescGPR(MemOp_STORE, FALSE, privileged, tagchecked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

address = GenerateAddress(address, offset, accdesc);

data = X[t, 8];
Mem[address, 1, accdesc] = data;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

C6.2.391 STURH

Store Register Halfword (unscaled) calculates an address from a base register value and an immediate offset, and stores a halfword to the calculated address, from a 32-bit register. For information about memory accesses, see *Load/store addressing modes*.

Encoding

```
STURH <Wt>, [<Xn|SP>{, #<simm>}]
```

Decode for this encoding

```
bits(64) offset = SignExtend(imm9, 64);
```

Assembler symbols

<Wt>	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<simm>	Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared decode for all encodings

```
integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tagchecked = n != 31;
```

Operation

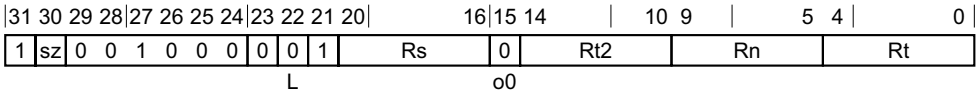
```
bits(64) address;
bits(16) data;

boolean privileged = PSTATE.EL != EL0;
AccessDescriptor accdesc = CreateAccDescGPR(MemOp_STORE, FALSE, privileged, tagchecked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

address = GenerateAddress(address, offset, accdesc);

data = X[t, 16];
Mem[address, 2, accdesc] = data;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

C6.2.392 STXP

Store Exclusive Pair of registers stores two 32-bit words or two 64-bit doublewords from two registers to a memory location if the PE has exclusive access to the memory address, and returns a status value of 0 if the store was successful, or of 1 if no store was performed. See *Synchronization and semaphores*. For information on single-copy atomicity and alignment requirements, see *Requirements for single-copy atomicity* and *Alignment of data accesses*. If a 64-bit pair Store-Exclusive succeeds, it causes a single-copy atomic update of the 128-bit memory location being updated. For information about memory accesses, see *Load/store addressing modes*.



32-bit variant

Applies when sz == 0.

```
STXP <Ws>, <Wt1>, <Wt2>, [<Xn|SP>{,#0}]
```

64-bit variant

Applies when sz == 1.

```
STXP <Ws>, <Xt1>, <Xt2>, [<Xn|SP>{,#0}]
```

Decode for all variants of this encoding

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2);    // ignored by load/store single register
integer s = UInt(Rs);    // ignored by all loads and store-release

constant integer elsize = 32 << UInt(sz);
constant integer datasize = elsize * 2;
boolean tagchecked = n != 31;

boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;
if s == t || (s == t2) then
    Constraint c = ConstrainUnpredictable(Unpredictable_DATAOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN rt_unknown = TRUE;    // store UNKNOWN value
        when Constraint_UNDEF   UNDEFINED;
        when Constraint_NOP     EndOfInstruction();
if s == n && n != 31 then
    Constraint c = ConstrainUnpredictable(Unpredictable_BASEOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN rn_unknown = TRUE;    // address is UNKNOWN
        when Constraint_UNDEF   UNDEFINED;
        when Constraint_NOP     EndOfInstruction();
```

Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*, and particularly *STXP*.

Assembler symbols

<Ws> Is the 32-bit name of the general-purpose register into which the status result of the store exclusive is written, encoded in the "Rs" field. The value returned is:

0 If the operation updates memory.

1 If the operation fails to update memory.

<Xt1> Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.

<Xt2> Is the 64-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.

<Wt1> Is the 32-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.

<Wt2> Is the 32-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Aborts and alignment

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- <Ws> is not updated.

Accessing an address that is not aligned to the size of the data being accessed causes an Alignment fault Data Abort exception to be generated, subject to the following rules:

- If AArch64.ExclusiveMonitorsPass() returns TRUE, the exception is generated.
- Otherwise, it is IMPLEMENTATION DEFINED whether the exception is generated.

If AArch64.ExclusiveMonitorsPass() returns FALSE and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.

Operation

```
bits(64) address;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;

AccessDescriptor accdesc = CreateAccDescExLDST(MemOp_STORE, FALSE, tagchecked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
elsif rn_unknown then
    address = bits(64) UNKNOWN;
else
    address = X[n, 64];

if rt_unknown then
    data = bits(datasize) UNKNOWN;
else
    bits(datasize DIV 2) el1 = X[t, datasize DIV 2];
    bits(datasize DIV 2) el2 = X[t2, datasize DIV 2];
    data = if BigEndian(accdesc.acctype) then el1:el2 else el2:el1;
bit status = '1';
// Check whether the Exclusives monitors are set to include the
// physical memory locations corresponding to virtual address
// range [address, address+dbytes-1].

// If AArch64.ExclusiveMonitorsPass() returns FALSE and the memory address,
// if accessed, would generate a synchronous Data Abort exception, it is
```

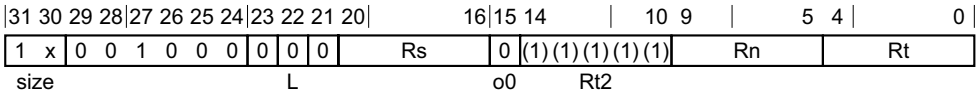
```
// IMPLEMENTATION DEFINED whether the exception is generated.
// It is a limitation of this model that synchronous Data Aborts are never
// generated in this case, as Mem[] is not called.
// If FEAT_SPE is implemented, it is also IMPLEMENTATION DEFINED whether or not the
// physical address packet is output when permitted and when
// AArch64.ExclusiveMonitorPass() returns FALSE for a Store Exclusive instruction.
// This behavior is not reflected here due to the previously stated limitation.
if AArch64.ExclusiveMonitorsPass(address, dbytes, accdesc) then
    // This atomic write will be rejected if it does not refer
    // to the same physical locations after address translation.
    Mem[address, dbytes, accdesc] = data;
    status = ExclusiveMonitorsStatus();
X[s, 32] = ZeroExtend(status, 32);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

C6.2.393 STXR

Store Exclusive Register stores a 32-bit word or a 64-bit doubleword from a register to memory if the PE has exclusive access to the memory address, and returns a status value of 0 if the store was successful, or of 1 if no store was performed. See *Synchronization and semaphores*. For information about memory accesses, see *Load/store addressing modes*.



32-bit variant

Applies when `size == 10`.

```
STXR <Ws>, <Wt>, [<Xn|SP>{,#0}]
```

64-bit variant

Applies when `size == 11`.

```
STXR <Ws>, <Xt>, [<Xn|SP>{,#0}]
```

Decode for all variants of this encoding

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer s = UInt(Rs);    // ignored by all loads and store-release

constant integer elsize = 8 << UInt(size);
boolean tagchecked = n != 31;

boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;
if s == t then
    Constraint c = ConstrainUnpredictable(Unpredictable_DATAOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN rt_unknown = TRUE;    // store UNKNOWN value
        when Constraint_UNDEF   UNDEFINED;
        when Constraint_NOP     EndOfInstruction();
if s == n && n != 31 then
    Constraint c = ConstrainUnpredictable(Unpredictable_BASEOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN rn_unknown = TRUE;    // address is UNKNOWN
        when Constraint_UNDEF   UNDEFINED;
        when Constraint_NOP     EndOfInstruction();
```

Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*, and particularly *STXR*.

Assembler symbols

`<Ws>` Is the 32-bit name of the general-purpose register into which the status result of the store exclusive is written, encoded in the "Rs" field. The value returned is:

`0` If the operation updates memory.

1 If the operation fails to update memory.

<Xt> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Aborts and alignment

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- <Ws> is not updated.

Accessing an address that is not aligned to the size of the data being accessed causes an Alignment fault Data Abort exception to be generated, subject to the following rules:

- If AArch64.ExclusiveMonitorsPass() returns TRUE, the exception is generated.
- Otherwise, it is IMPLEMENTATION DEFINED whether the exception is generated.

If AArch64.ExclusiveMonitorsPass() returns FALSE and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.

Operation

```
bits(64) address;
bits(elsize) data;
constant integer dbytes = elsize DIV 8;

AccessDescriptor accdesc = CreateAccDescExLDST(MemOp_STORE, FALSE, tagchecked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
elsif rn_unknown then
    address = bits(64) UNKNOWN;
else
    address = X[n, 64];

if rt_unknown then
    data = bits(elsize) UNKNOWN;
else
    data = X[t, elsize];

bit status = '1';
// Check whether the Exclusives monitors are set to include the
// physical memory locations corresponding to virtual address
// range [address, address+dbytes-1].

// If AArch64.ExclusiveMonitorsPass() returns FALSE and the memory address,
// if accessed, would generate a synchronous Data Abort exception, it is
// IMPLEMENTATION DEFINED whether the exception is generated.
// It is a limitation of this model that synchronous Data Aborts are never
// generated in this case, as Mem[] is not called.
// If FEAT_SPE is implemented, it is also IMPLEMENTATION DEFINED whether or not the
// physical address packet is output when permitted and when
// AArch64.ExclusiveMonitorPass() returns FALSE for a Store Exclusive instruction.
// This behavior is not reflected here due to the previously stated limitation.
if AArch64.ExclusiveMonitorsPass(address, dbytes, accdesc) then
    // This atomic write will be rejected if it does not refer
    // to the same physical locations after address translation.
    Mem[address, dbytes, accdesc] = data;
```

```
    status = ExclusiveMonitorsStatus();
X[s, 32] = ZeroExtend(status, 32);
```

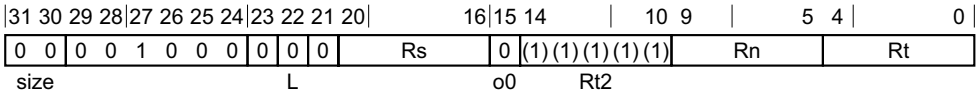
Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

C6.2.394 STXRB

Store Exclusive Register Byte stores a byte from a register to memory if the PE has exclusive access to the memory address, and returns a status value of 0 if the store was successful, or of 1 if no store was performed. See *Synchronization and semaphores*. The memory access is atomic.

For information about memory accesses, see *Load/store addressing modes*.



Encoding

```
STXRB <Ws>, <Wt>, [<Xn|SP>{,#0}]
```

Decode for this encoding

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer s = UInt(Rs);    // ignored by all loads and store-release

boolean tagchecked = n != 31;

boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;
if s == t then
    Constraint c = ConstrainUnpredictable(Unpredictable_DATAOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN rt_unknown = TRUE;    // store UNKNOWN value
        when Constraint_UNDEF   UNDEFINED;
        when Constraint_NOP     EndOfInstruction();
if s == n && n != 31 then
    Constraint c = ConstrainUnpredictable(Unpredictable_BASEOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN rn_unknown = TRUE;    // address is UNKNOWN
        when Constraint_UNDEF   UNDEFINED;
        when Constraint_NOP     EndOfInstruction();
```

Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*, and particularly *STXRB*.

Assembler symbols

`<Ws>` Is the 32-bit name of the general-purpose register into which the status result of the store exclusive is written, encoded in the "Rs" field. The value returned is:

- `0` If the operation updates memory.
- `1` If the operation fails to update memory.

`<Wt>` Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

`<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Aborts

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- <Ws> is not updated.

If AArch64.ExclusiveMonitorsPass() returns FALSE and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.

Operation

```
bits(64) address;
bits(8) data;

AccessDescriptor accdesc = CreateAccDescExLDST(MemOp_STORE, FALSE, tagchecked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
elsif rn_unknown then
    address = bits(64) UNKNOWN;
else
    address = X[n, 64];

if rt_unknown then
    data = bits(8) UNKNOWN;
else
    data = X[t, 8];

bit status = '1';
// Check whether the Exclusives monitors are set to include the
// physical memory locations corresponding to virtual address
// range [address, address+dbytes-1].

// If AArch64.ExclusiveMonitorsPass() returns FALSE and the memory address,
// if accessed, would generate a synchronous Data Abort exception, it is
// IMPLEMENTATION DEFINED whether the exception is generated.
// It is a limitation of this model that synchronous Data Aborts are never
// generated in this case, as Mem[] is not called.
// If FEAT_SPE is implemented, it is also IMPLEMENTATION DEFINED whether or not the
// physical address packet is output when permitted and when
// AArch64.ExclusiveMonitorPass() returns FALSE for a Store Exclusive instruction.
// This behavior is not reflected here due to the previously stated limitation.
if AArch64.ExclusiveMonitorsPass(address, 1, accdesc) then
    // This atomic write will be rejected if it does not refer
    // to the same physical locations after address translation.
    Mem[address, 1, accdesc] = data;
    status = ExclusiveMonitorsStatus();
X[s, 32] = ZeroExtend(status, 32);
```

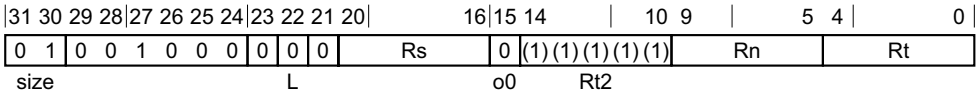
Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

C6.2.395 STXRH

Store Exclusive Register Halfword stores a halfword from a register to memory if the PE has exclusive access to the memory address, and returns a status value of 0 if the store was successful, or of 1 if no store was performed. See *Synchronization and semaphores*. The memory access is atomic.

For information about memory accesses, see *Load/store addressing modes*.



Encoding

```
STXRH <Ws>, <Wt>, [<Xn|SP>{,#0}]
```

Decode for this encoding

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer s = UInt(Rs);    // ignored by all loads and store-release

boolean tagchecked = n != 31;

boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;
if s == t then
    Constraint c = ConstrainUnpredictable(Unpredictable_DATAOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN rt_unknown = TRUE;    // store UNKNOWN value
        when Constraint_UNDEF   UNDEFINED;
        when Constraint_NOP     EndOfInstruction();
if s == n && n != 31 then
    Constraint c = ConstrainUnpredictable(Unpredictable_BASEOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN rn_unknown = TRUE;    // address is UNKNOWN
        when Constraint_UNDEF   UNDEFINED;
        when Constraint_NOP     EndOfInstruction();
```

Assembler symbols

`<Ws>` Is the 32-bit name of the general-purpose register into which the status result of the store exclusive is written, encoded in the "Rs" field. The value returned is:

- `0` If the operation updates memory.
- `1` If the operation fails to update memory.

`<Wt>` Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

`<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Aborts and alignment

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- `<Ws>` is not updated.

A non halfword-aligned memory address causes an Alignment fault Data Abort exception to be generated, subject to the following rules:

- If AArch64.ExclusiveMonitorsPass() returns TRUE, the exception is generated.
- Otherwise, it is IMPLEMENTATION DEFINED whether the exception is generated.

If AArch64.ExclusiveMonitorsPass() returns FALSE and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.

Operation

```
bits(64) address;
bits(16) data;

AccessDescriptor accdesc = CreateAccDescExLDST(MemOp_STORE, FALSE, tagchecked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
elsif rn_unknown then
    address = bits(64) UNKNOWN;
else
    address = X[n, 64];

if rt_unknown then
    data = bits(16) UNKNOWN;
else
    data = X[t, 16];

bit status = '1';
// Check whether the Exclusives monitors are set to include the
// physical memory locations corresponding to virtual address
// range [address, address+dbytes-1].

// If AArch64.ExclusiveMonitorsPass() returns FALSE and the memory address,
// if accessed, would generate a synchronous Data Abort exception, it is
// IMPLEMENTATION DEFINED whether the exception is generated.
// It is a limitation of this model that synchronous Data Aborts are never
// generated in this case, as Mem[] is not called.
// If FEAT_SPE is implemented, it is also IMPLEMENTATION DEFINED whether or not the
// physical address packet is output when permitted and when
// AArch64.ExclusiveMonitorPass() returns FALSE for a Store Exclusive instruction.
// This behavior is not reflected here due to the previously stated limitation.
if AArch64.ExclusiveMonitorsPass(address, 2, accdesc) then
    // This atomic write will be rejected if it does not refer
    // to the same physical locations after address translation.
    Mem[address, 2, accdesc] = data;
    status = ExclusiveMonitorsStatus();
X[s, 32] = ZeroExtend(status, 32);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

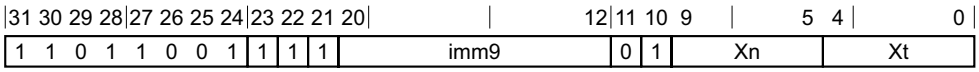
C6.2.396 STZ2G

Store Allocation Tags, Zeroing stores an Allocation Tag to two Tag granules of memory, zeroing the associated data locations. The address used for the store is calculated from the base register and an immediate signed offset scaled by the Tag granule. The Allocation Tag is calculated from the Logical Address Tag in the source register.

This instruction generates an Unchecked access.

Post-index

(FEAT_MTE)



Encoding

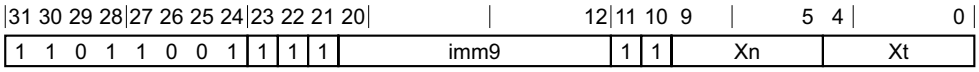
```
STZ2G <Xt|SP>, [<Xn|SP>], #<simm>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_MTE) then UNDEFINED;
integer n = UInt(Xn);
integer t = UInt(Xt);
bits(64) offset = LSL(SignExtend(imm9, 64), LOG2_TAG_GRANULE);
boolean writeback = TRUE;
boolean postindex = TRUE;
```

Pre-index

(FEAT_MTE)



Encoding

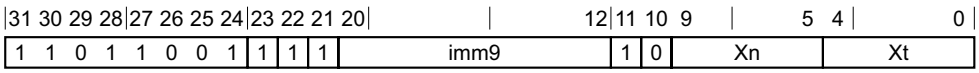
```
STZ2G <Xt|SP>, [<Xn|SP>, #<simm>]!
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_MTE) then UNDEFINED;
integer n = UInt(Xn);
integer t = UInt(Xt);
bits(64) offset = LSL(SignExtend(imm9, 64), LOG2_TAG_GRANULE);
boolean writeback = TRUE;
boolean postindex = FALSE;
```

Signed offset

(FEAT_MTE)



Encoding

```
STZ2G <Xt|SP>, [<Xn|SP>{, #<simm>}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_MTE) then UNDEFINED;
integer n = UInt(Xn);
integer t = UInt(Xt);
bits(64) offset = LSL(SignExtend(imm9, 64), LOG2_TAG_GRANULE);
boolean writeback = FALSE;
boolean postindex = FALSE;
```

Assembler symbols

<Xt|SP>	Is the 64-bit name of the general-purpose source register or stack pointer, encoded in the "Xt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Xn" field.
<simm>	Is the optional signed immediate offset, a multiple of 16 in the range -4096 to 4080, defaulting to 0 and encoded in the "imm9" field.

Operation for all encodings

```
bits(64) address;
bits(64) address2;
bits(64) data = if t == 31 then SP[] else X[t, 64];
bits(4) tag = AArch64.AllocationTagFromAddress(data);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

AccessDescriptor accdesc = CreateAccDescLDGSTG(MemOp_STORE);

if !postindex then
    address = GenerateAddress(address, offset, accdesc);

address2 = GenerateAddress(address, TAG_GRANULE, accdesc);

if !IsAligned(address, TAG_GRANULE) then
    AArch64.Abort(address, AlignmentFault(accdesc));

Mem[address, TAG_GRANULE, accdesc] = Zeros(TAG_GRANULE * 8);
Mem[address2, TAG_GRANULE, accdesc] = Zeros(TAG_GRANULE * 8);

AArch64.MemTag[address, accdesc] = tag;
AArch64.MemTag[address2, accdesc] = tag;

if writeback then
    if postindex then
        address = GenerateAddress(address, offset, accdesc);

    if n == 31 then
        SP[] = address;
    else
        X[n, 64] = address;
```

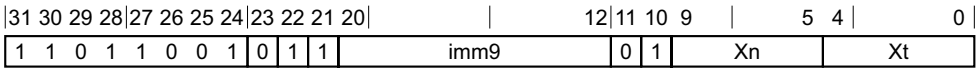
C6.2.397 STZG

Store Allocation Tag, Zeroing stores an Allocation Tag to memory, zeroing the associated data location. The address used for the store is calculated from the base register and an immediate signed offset scaled by the Tag granule. The Allocation Tag is calculated from the Logical Address Tag in the source register.

This instruction generates an Unchecked access.

Post-index

(FEAT_MTE)



Encoding

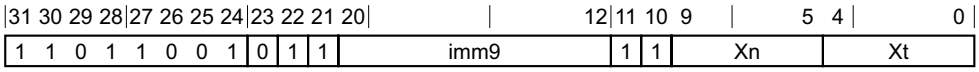
```
STZG <Xt|SP>, [<Xn|SP>], #<simm>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_MTE) then UNDEFINED;
integer n = UInt(Xn);
integer t = UInt(Xt);
bits(64) offset = LSL(SignExtend(imm9, 64), LOG2_TAG_GRANULE);
boolean writeback = TRUE;
boolean postindex = TRUE;
```

Pre-index

(FEAT_MTE)



Encoding

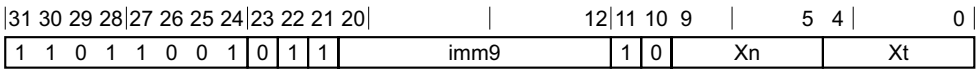
```
STZG <Xt|SP>, [<Xn|SP>, #<simm>]!
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_MTE) then UNDEFINED;
integer n = UInt(Xn);
integer t = UInt(Xt);
bits(64) offset = LSL(SignExtend(imm9, 64), LOG2_TAG_GRANULE);
boolean writeback = TRUE;
boolean postindex = FALSE;
```

Signed offset

(FEAT_MTE)



Encoding

```
STZG <Xt|SP>, [<Xn|SP>{, #<simm>}]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_MTE) then UNDEFINED;
integer n = UInt(Xn);
integer t = UInt(Xt);
bits(64) offset = LSL(SignExtend(imm9, 64), LOG2_TAG_GRANULE);
boolean writeback = FALSE;
boolean postindex = FALSE;
```

Assembler symbols

<Xt|SP>	Is the 64-bit name of the general-purpose source register or stack pointer, encoded in the "Xt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Xn" field.
<simm>	Is the optional signed immediate offset, a multiple of 16 in the range -4096 to 4080, defaulting to 0 and encoded in the "imm9" field.

Operation for all encodings

```
bits(64) address;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

AccessDescriptor accdesc = CreateAccDescLDGSTG(MemOp_STORE);

if !postindex then
    address = GenerateAddress(address, offset, accdesc);

if !IsAligned(address, TAG_GRANULE) then
    AArch64.Abort(address, AlignmentFault(accdesc));

Mem[address, TAG_GRANULE, accdesc] = Zeros(TAG_GRANULE * 8);

bits(64) data = if t == 31 then SP[] else X[t, 64];
bits(4) tag = AArch64.AllocationTagFromAddress(data);
AArch64.MemTag[address, accdesc] = tag;

if writeback then
    if postindex then
        address = GenerateAddress(address, offset, accdesc);

    if n == 31 then
        SP[] = address;
    else
        X[n, 64] = address;
```

C6.2.398 STZGM

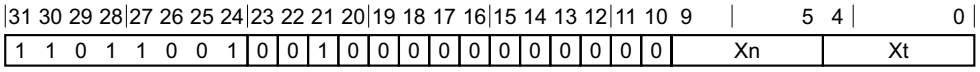
Store Tag and Zero Multiple writes a naturally aligned block of N Allocation Tags and stores zero to the associated data locations, where the size of N is identified in DCZID_EL0.BS, and the Allocation Tag is taken from the source register bits<3:0>.

This instruction is UNDEFINED at EL0.

This instruction generates an Unchecked access.

Integer

(FEAT_MTE2)



Encoding

```
STZGM <Xt>, [<Xn|SP>]
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_MTE2) then UNDEFINED;
integer t = UInt(Xt);
integer n = UInt(Xn);
```

Assembler symbols

<Xt> Is the 64-bit name of the general-purpose source register, encoded in the "Xt" field.

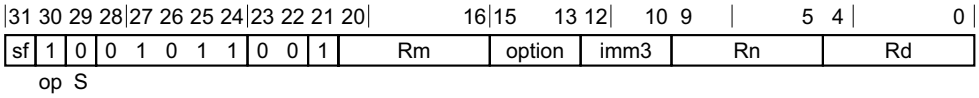
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Xn" field.

Operation

```
if PSTATE.EL == EL0 then
    UNDEFINED;

bits(64) data = X[t, 64];
bits(4) tag = data<3:0>;
bits(64) address;
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

integer size = 4 * (2 ^ (UInt(DCZID_EL0.BS)));
address = Align(address, size);
integer count = size >> LOG2_TAG_GRANULE;
AccessDescriptor accdesc = CreateAccDescLDGSTG(MemOp_STORE);

for i = 0 to count-1
    AArch64.MemTag[address, accdesc] = tag;
    Mem[address, TAG_GRANULE, accdesc] = Zeros(8 * TAG_GRANULE);
    address = GenerateAddress(address, TAG_GRANULE, accdesc);
```

C6.2.399 SUB (extended register)

Subtract (extended register) subtracts a sign or zero-extended register value, followed by an optional left shift amount, from a register value, and writes the result to the destination register. The argument that is extended from the <Rm> register can be a byte, halfword, word, or doubleword.



32-bit variant

Applies when sf == 0.

```
SUB <Wd|WSP>, <Wn|WSP>, <Wm>{, <extend> {#<amount>}}
```

64-bit variant

Applies when sf == 1.

```
SUB <Xd|SP>, <Xn|SP>, <R><m>{, <extend> {#<amount>}}
```

Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
constant integer datasize = 32 << UInt(sf);
ExtendType extend_type = DecodeRegExtend(option);
integer shift = UInt(imm3);
if shift > 4 then UNDEFINED;
```

Assembler symbols

<Wd|WSP> Is the 32-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.

<Wn|WSP> Is the 32-bit name of the first source general-purpose register or stack pointer, encoded in the "Rn" field.

<Wm> Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.

<Xd|SP> Is the 64-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.

<Xn|SP> Is the 64-bit name of the first source general-purpose register or stack pointer, encoded in the "Rn" field.

<R> Is a width specifier, encoded in the "option" field. It can have the following values:

W	when option = 00x
W	when option = 010
X	when option = x11
W	when option = 10x
W	when option = 110

<m> Is the number [0-30] of the second general-purpose source register or the name ZR (31), encoded in the "Rm" field.

<extend> For the 32-bit variant: is the extension to be applied to the second source operand, encoded in the "option" field. It can have the following values:

UXTB when option = 000

UXTH when option = 001

LSL|UXTW when option = 010

UXTX when option = 011

SXTB when option = 100

SXTH when option = 101

SXTW when option = 110

SXTX when option = 111

If "Rd" or "Rn" is '11111' (WSP) and "option" is '010' then LSL is preferred, but may be omitted when "imm3" is '000'. In all other cases <extend> is required and must be UXTW when "option" is '010'.

For the 64-bit variant: is the extension to be applied to the second source operand, encoded in the "option" field. It can have the following values:

UXTB when option = 000

UXTH when option = 001

UXTW when option = 010

LSL|UXTX when option = 011

SXTB when option = 100

SXTH when option = 101

SXTW when option = 110

SXTX when option = 111

If "Rd" or "Rn" is '11111' (SP) and "option" is '011' then LSL is preferred, but may be omitted when "imm3" is '000'. In all other cases <extend> is required and must be UXTX when "option" is '011'.

<amount> Is the left shift amount to be applied after extension in the range 0 to 4, defaulting to 0, encoded in the "imm3" field. It must be absent when <extend> is absent, is required when <extend> is LSL, and is optional when <extend> is present but not LSL.

Operation

```
bits(datasize) result;
bits(datasize) operand1 = if n == 31 then SP[]<datasize-1:0> else X[n, datasize];
bits(datasize) operand2 = ExtendReg(m, extend_type, shift, datasize);

operand2 = NOT(operand2);
(result, -) = AddWithCarry(operand1, operand2, '1');

if d == 31 then
    SP[] = ZeroExtend(result, 64);
else
    X[d, datasize] = result;
```

Operational information

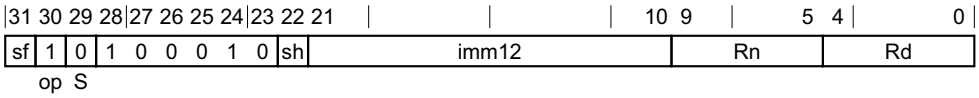
If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

C6.2.400 SUB (immediate)

Subtract (immediate) subtracts an optionally-shifted immediate value from a register value, and writes the result to the destination register.



32-bit variant

Applies when sf == 0.

```
SUB <Wd|WSP>, <Wn|WSP>, #<imm>{, <shift>}
```

64-bit variant

Applies when sf == 1.

```
SUB <Xd|SP>, <Xn|SP>, #<imm>{, <shift>}
```

Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
constant integer datasize = 32 << UInt(sf);
bits(datasize) imm;

case sh of
    when '0' imm = ZeroExtend(imm12, datasize);
    when '1' imm = ZeroExtend(imm12:Zeros(12), datasize);
```

Assembler symbols

<Wd|WSP> Is the 32-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.

<Wn|WSP> Is the 32-bit name of the source general-purpose register or stack pointer, encoded in the "Rn" field.

<Xd|SP> Is the 64-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.

<Xn|SP> Is the 64-bit name of the source general-purpose register or stack pointer, encoded in the "Rn" field.

<imm> Is an unsigned immediate, in the range 0 to 4095, encoded in the "imm12" field.

<shift> Is the optional left shift to apply to the immediate, defaulting to LSL #0 and encoded in the "sh" field. It can have the following values:

LSL #0 when sh = 0

LSL #12 when sh = 1

Operation

```
bits(datasize) result;
bits(datasize) operand1 = if n == 31 then SP[]<datasize-1:0> else X[n, datasize];
bits(datasize) operand2;

operand2 = NOT(imm);
(result, -) = AddWithCarry(operand1, operand2, '1');
```

```
if d == 31 then
    SP[] = ZeroExtend(result, 64);
else
    X[d, datasize] = result;
```

Operational information

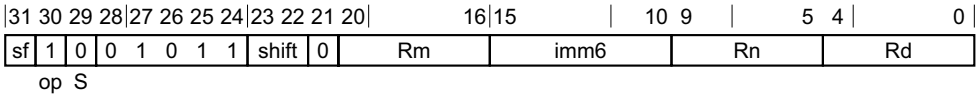
If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

C6.2.401 SUB (shifted register)

Subtract (shifted register) subtracts an optionally-shifted register value from a register value, and writes the result to the destination register.

This instruction is used by the alias NEG (shifted register). See *Alias conditions* for details of when each alias is preferred.



32-bit variant

Applies when `sf == 0`.

```
SUB <Wd>, <Wn>, <Wm>{, <shift> #<amount>}
```

64-bit variant

Applies when `sf == 1`.

```
SUB <Xd>, <Xn>, <Xm>{, <shift> #<amount>}
```

Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
constant integer datasize = 32 << UInt(sf);

if shift == '11' then UNDEFINED;
if sf == '0' && imm6<5> == '1' then UNDEFINED;

ShiftType shift_type = DecodeShift(shift);
integer shift_amount = UInt(imm6);
```

Alias conditions

Alias	is preferred when
NEG (shifted register)	`Rn == '11111'`

Assembler symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Wm>	Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.

<shift> Is the optional shift type to be applied to the second source operand, defaulting to LSL and encoded in the "shift" field. It can have the following values:

LSL when shift = 00

LSR when shift = 01

ASR when shift = 10

The encoding shift = 11 is reserved.

<amount> For the 32-bit variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field.

For the 64-bit variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field.

Operation

```
bits(datasize) result;
bits(datasize) operand1 = X[n, datasize];
bits(datasize) operand2 = ShiftReg(m, shift_type, shift_amount, datasize);

operand2 = NOT(operand2);
(result, -) = AddWithCarry(operand1, operand2, '1');

X[d, datasize] = result;
```

Operational information

If PSTATE.DIT is 1:

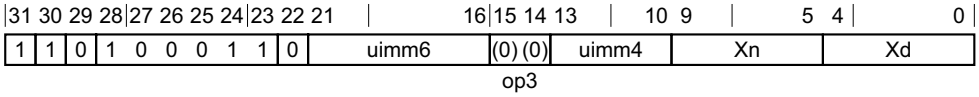
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

C6.2.402 SUBG

Subtract with Tag subtracts an immediate value scaled by the Tag granule from the address in the source register, modifies the Logical Address Tag of the address using an immediate value, and writes the result to the destination register. Tags specified in GCR_EL1.Exclude are excluded from the possible outputs when modifying the Logical Address Tag.

Integer

(FEAT_MTE)



Encoding

```
SUBG <Xd|SP>, <Xn|SP>, #<uimm6>, #<uimm4>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_MTE) then UNDEFINED;
integer d = UInt(Xd);
integer n = UInt(Xn);
bits(64) offset = LSL(ZeroExtend(uimm6, 64), LOG2_TAG_GRANULE);
```

Assembler symbols

<Xd|SP> Is the 64-bit name of the destination general-purpose register or stack pointer, encoded in the "Xd" field.

<Xn|SP> Is the 64-bit name of the source general-purpose register or stack pointer, encoded in the "Xn" field.

<uimm6> Is an unsigned immediate, a multiple of 16 in the range 0 to 1008, encoded in the "uimm6" field.

<uimm4> Is an unsigned immediate, in the range 0 to 15, encoded in the "uimm4" field.

Operation

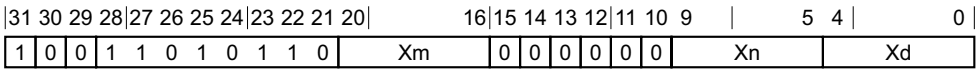
```
bits(64) operand1 = if n == 31 then SP[] else X[n, 64];
bits(4) start_tag = AArch64.AllocationTagFromAddress(operand1);
bits(16) exclude = GCR_EL1.Exclude;
bits(64) result;
bits(4) rtag;

if AArch64.AllocationTagAccessIsEnabled(PSTATE.EL) then
    rtag = AArch64.ChooseNonExcludedTag(start_tag, uimm4, exclude);
else
    rtag = '0000';

(result, -) = AddWithCarry(operand1, NOT(offset), '1');

result = AArch64.AddressWithAllocationTag(result, rtag);

if d == 31 then
    SP[] = result;
else
    X[d, 64] = result;
```

C6.2.403 SUBP

Subtract Pointer subtracts the 56-bit address held in the second source register from the 56-bit address held in the first source register, sign-extends the result to 64-bits, and writes the result to the destination register.

Integer

(FEAT_MTE)



Encoding

```
SUBP <Xd>, <Xn|SP>, <Xm|SP>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_MTE) then UNDEFINED;
integer d = UInt(Xd);
integer n = UInt(Xn);
integer m = UInt(Xm);
```

Assembler symbols

<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Xd" field.
<Xn|SP>	Is the 64-bit name of the first source general-purpose register or stack pointer, encoded in the "Xn" field.
<Xm|SP>	Is the 64-bit name of the second general-purpose source register or stack pointer, encoded in the "Xm" field.

Operation

```
bits(64) operand1 = if n == 31 then SP[] else X[n, 64];
bits(64) operand2 = if m == 31 then SP[] else X[m, 64];
operand1 = SignExtend(operand1<55:0>, 64);
operand2 = SignExtend(operand2<55:0>, 64);

bits(64) result;

operand2 = NOT(operand2);
(result, -) = AddWithCarry(operand1, operand2, '1');

X[d, 64] = result;
```

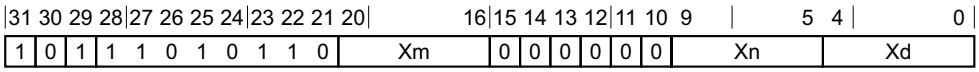
C6.2.404 SUBPS

Subtract Pointer, setting Flags subtracts the 56-bit address held in the second source register from the 56-bit address held in the first source register, sign-extends the result to 64-bits, and writes the result to the destination register. It updates the condition flags based on the result of the subtraction.

This instruction is used by the alias CMPP. See *Alias conditions* for details of when each alias is preferred.

Integer

(FEAT_MTE)



Encoding

```
SUBPS <Xd>, <Xn|SP>, <Xm|SP>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_MTE) then UNDEFINED;
integer d = UInt(Xd);
integer n = UInt(Xn);
integer m = UInt(Xm);
```

Alias conditions

Alias	is preferred when
CMPP	`S == '1' && Xd == '11111'`

Assembler symbols

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Xd" field.

<Xn|SP> Is the 64-bit name of the first source general-purpose register or stack pointer, encoded in the "Xn" field.

<Xm|SP> Is the 64-bit name of the second general-purpose source register or stack pointer, encoded in the "Xm" field.

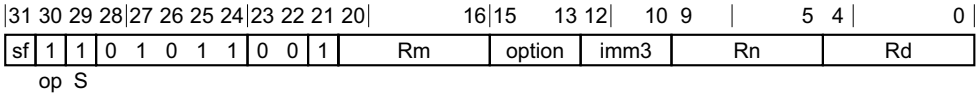
Operation

```
bits(64) operand1 = if n == 31 then SP[] else X[n, 64];
bits(64) operand2 = if m == 31 then SP[] else X[m, 64];
operand1 = SignExtend(operand1<55:0>, 64);
operand2 = SignExtend(operand2<55:0>, 64);

bits(64) result;
bits(4) nzcv;

operand2 = NOT(operand2);
(result, nzcv) = AddWithCarry(operand1, operand2, '1');

PSTATE.<N,Z,C,V> = nzcv;
X[d, 64] = result;
```

C6.2.405 SUBS (extended register)

Subtract (extended register), setting flags, subtracts a sign or zero-extended register value, followed by an optional left shift amount, from a register value, and writes the result to the destination register. The argument that is extended from the <Rm> register can be a byte, halfword, word, or doubleword. It updates the condition flags based on the result.

This instruction is used by the alias CMP (extended register). See Alias conditions for details of when each alias is preferred.



32-bit variant

Applies when sf == 0.

```
SUBS <Wd>, <Wn|WSP>, <Wm>{, <extend> {#<amount>}}
```

64-bit variant

Applies when sf == 1.

```
SUBS <Xd>, <Xn|SP>, <R><m>{, <extend> {#<amount>}}
```

Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
constant integer datasize = 32 << UInt(sf);
ExtendType extend_type = DecodeRegExtend(option);
integer shift = UInt(imm3);
if shift > 4 then UNDEFINED;
```

Alias conditions

Alias	is preferred when
CMP (extended register)	Rd == '11111'

Assembler symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Wn|WSP> Is the 32-bit name of the first source general-purpose register or stack pointer, encoded in the "Rn" field.

<Wm> Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Xn|SP> Is the 64-bit name of the first source general-purpose register or stack pointer, encoded in the "Rn" field.

<R> Is a width specifier, encoded in the "option" field. It can have the following values:

W when option = 00x

W when option = 010

X when option = x11

W when option = 10x

W when option = 110

<m> Is the number [0-30] of the second general-purpose source register or the name ZR (31), encoded in the "Rm" field.

<extend> For the 32-bit variant: is the extension to be applied to the second source operand, encoded in the "option" field. It can have the following values:

UXTB when option = 000

UXTH when option = 001

LSL|UXTW when option = 010

UXTX when option = 011

SXTB when option = 100

SXTH when option = 101

SXTW when option = 110

SXTX when option = 111

If "Rn" is '11111' (WSP) and "option" is '010' then LSL is preferred, but may be omitted when "imm3" is '000'. In all other cases <extend> is required and must be UXTW when "option" is '010'.

For the 64-bit variant: is the extension to be applied to the second source operand, encoded in the "option" field. It can have the following values:

UXTB when option = 000

UXTH when option = 001

UXTW when option = 010

LSL|UXTX when option = 011

SXTB when option = 100

SXTH when option = 101

SXTW when option = 110

SXTX when option = 111

If "Rn" is '11111' (SP) and "option" is '011' then LSL is preferred, but may be omitted when "imm3" is '000'. In all other cases <extend> is required and must be UXTX when "option" is '011'.

<amount> Is the left shift amount to be applied after extension in the range 0 to 4, defaulting to 0, encoded in the "imm3" field. It must be absent when <extend> is absent, is required when <extend> is LSL, and is optional when <extend> is present but not LSL.

Operation

```
bits(datasize) result;
bits(datasize) operand1 = if n == 31 then SP[]<datasize-1:0> else X[n, datasize];
bits(datasize) operand2 = ExtendReg(m, extend_type, shift, datasize);
bits(4) nzcv;

operand2 = NOT(operand2);
(result, nzcv) = AddWithCarry(operand1, operand2, '1');

PSTATE.<N,Z,C,V> = nzcv;

X[d, datasize] = result;
```

Operational information

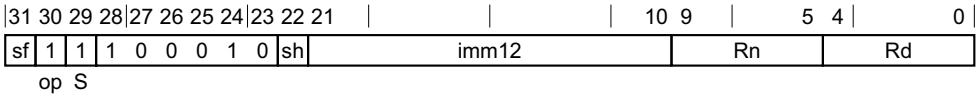
If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

C6.2.406 SUBS (immediate)

Subtract (immediate), setting flags, subtracts an optionally-shifted immediate value from a register value, and writes the result to the destination register. It updates the condition flags based on the result.

This instruction is used by the alias CMP (immediate). See *Alias conditions* for details of when each alias is preferred.



32-bit variant

Applies when `sf == 0`.

```
SUBS <Wd>, <Wn|WSP>, #<imm>{, <shift>}
```

64-bit variant

Applies when `sf == 1`.

```
SUBS <Xd>, <Xn|SP>, #<imm>{, <shift>}
```

Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
constant integer datasize = 32 << UInt(sf);
bits(datasize) imm;

case sh of
    when '0' imm = ZeroExtend(imm12, datasize);
    when '1' imm = ZeroExtend(imm12:Zeros(12), datasize);
```

Alias conditions

Alias	is preferred when
CMP (immediate)	`Rd == '11111'`

Assembler symbols

`<Wd>` Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

`<Wn|WSP>` Is the 32-bit name of the source general-purpose register or stack pointer, encoded in the "Rn" field.

`<Xd>` Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

`<Xn|SP>` Is the 64-bit name of the source general-purpose register or stack pointer, encoded in the "Rn" field.

`<imm>` Is an unsigned immediate, in the range 0 to 4095, encoded in the "imm12" field.

`<shift>` Is the optional left shift to apply to the immediate, defaulting to LSL #0 and encoded in the "sh" field. It can have the following values:

`LSL #0` when `sh = 0`

`LSL #12` when `sh = 1`

Operation

```
bits(datasize) result;
bits(datasize) operand1 = if n == 31 then SP[]<datasize-1:0> else X[n, datasize];
bits(datasize) operand2;
bits(4) nzcv;

operand2 = NOT(imm);
(result, nzcv) = AddWithCarry(operand1, operand2, '1');

PSTATE.<N,Z,C,V> = nzcv;

X[d, datasize] = result;
```

Operational information

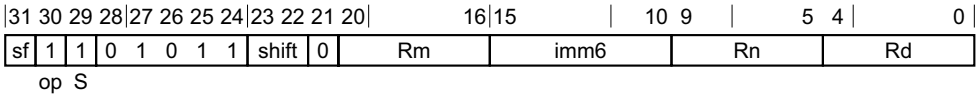
If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

C6.2.407 SUBS (shifted register)

Subtract (shifted register), setting flags, subtracts an optionally-shifted register value from a register value, and writes the result to the destination register. It updates the condition flags based on the result.

This instruction is used by the aliases CMP (shifted register) and NEGS. See *Alias conditions* for details of when each alias is preferred.



32-bit variant

Applies when sf == 0.

```
SUBS <Wd>, <Wn>, <Wm>{, <shift> #<amount>}
```

64-bit variant

Applies when sf == 1.

```
SUBS <Xd>, <Xn>, <Xm>{, <shift> #<amount>}
```

Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
constant integer datasize = 32 << UInt(sf);

if shift == '11' then UNDEFINED;
if sf == '0' && imm6<5> == '1' then UNDEFINED;

ShiftType shift_type = DecodeShift(shift);
integer shift_amount = UInt(imm6);
```

Alias conditions

Alias	is preferred when
CMP (shifted register)	Rd == '11111'
NEGS	Rn == '11111' && Rd != '11111'

Assembler symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Wn> Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.

<Wm> Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Xn> Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.

<Xm> Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.

<shift> Is the optional shift type to be applied to the second source operand, defaulting to LSL and encoded in the "shift" field. It can have the following values:

LSL when shift = 00

LSR when shift = 01

ASR when shift = 10

The encoding shift = 11 is reserved.

<amount> For the 32-bit variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field.

For the 64-bit variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field.

Operation

```
bits(datasize) result;
bits(datasize) operand1 = X[n, datasize];
bits(datasize) operand2 = ShiftReg(m, shift_type, shift_amount, datasize);
bits(4) nzcv;

operand2 = NOT(operand2);
(result, nzcv) = AddWithCarry(operand1, operand2, '1');

PSTATE.<N,Z,C,V> = nzcv;

X[d, datasize] = result;
```

Operational information

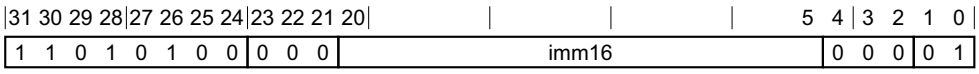
If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

C6.2.408 SVC

Supervisor Call causes an exception to be taken to EL1.

On executing an SVC instruction, the PE records the exception as a Supervisor Call exception in ESR_ELx, using the EC value 0x15, and the value of the immediate argument.



Encoding

SVC #<imm>

Decode for this encoding

```
// Empty.
```

Assembler symbols

<imm> Is a 16-bit unsigned immediate, in the range 0 to 65535, encoded in the "imm16" field.

Operation

```
AArch64.CheckForSVCTrap(imm16);
AArch64.CallSupervisor(imm16);
```

C6.2.409 SWP, SWPA, SWPAL, SWPL

Swap word or doubleword in memory atomically loads a 32-bit word or 64-bit doubleword from a memory location, and stores the value held in a register back to the same memory location. The value initially loaded from memory is returned in the destination register.

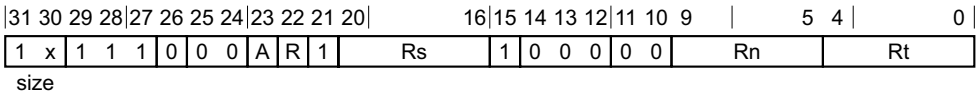
- If the destination register is not one of `WZR` or `XZR`, `SWPA` and `SWPAL` load from memory with acquire semantics.
- `SWPL` and `SWPAL` store to memory with release semantics.
- `SWP` has neither acquire nor release semantics.

For more information about memory ordering semantics, see *Load-Acquire, Load-AcquirePC, and Store-Release*.

For information about memory accesses, see *Load/store addressing modes*.

Integer

(FEAT_LSE)



32-bit SWP variant

Applies when `size == 10 && A == 0 && R == 0`.

```
SWP <Ws>, <Wt>, [<Xn|SP>]
```

32-bit SWPA variant

Applies when `size == 10 && A == 1 && R == 0`.

```
SWPA <Ws>, <Wt>, [<Xn|SP>]
```

32-bit SWPAL variant

Applies when `size == 10 && A == 1 && R == 1`.

```
SWPAL <Ws>, <Wt>, [<Xn|SP>]
```

32-bit SWPL variant

Applies when `size == 10 && A == 0 && R == 1`.

```
SWPL <Ws>, <Wt>, [<Xn|SP>]
```

64-bit SWP variant

Applies when `size == 11 && A == 0 && R == 0`.

```
SWP <Xs>, <Xt>, [<Xn|SP>]
```

64-bit SWPA variant

Applies when `size == 11 && A == 1 && R == 0`.

```
SWPA <Xs>, <Xt>, [<Xn|SP>]
```

64-bit SWPAL variant

Applies when `size == 11 && A == 1 && R == 1`.

```
SWPAL <Xs>, <Xt>, [<Xn|SP>]
```

64-bit SWPL variant

Applies when size == 11 && A == 0 && R == 1.

```
SWPL <Xs>, <Xt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LSE) then UNDEFINED;

integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

constant integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
boolean acquire = A == '1' && Rt != '11111';
boolean release = R == '1';
boolean tagchecked = n != 31;
```

Assembler symbols

<Ws>	Is the 32-bit name of the general-purpose register to be stored, encoded in the "Rs" field.
<Wt>	Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xs>	Is the 64-bit name of the general-purpose register to be stored, encoded in the "Rs" field.
<Xt>	Is the 64-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(datasize) data;
bits(datasize) store_value;
AccessDescriptor accdesc = CreateAccDescAtomicOp(MemAtomicOp_SWP, acquire, release, tagchecked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

store_value = X[s, datasize];

bits(datasize) comparevalue = bits(datasize) UNKNOWN;    // Irrelevant when not executing CAS
data = MemAtomic(address, comparevalue, store_value, accdesc);

X[t, regsize] = ZeroExtend(data, regsize);
```

C6.2.410 SWPB, SWPAB, SWPALB, SWPLB

Swap byte in memory atomically loads an 8-bit byte from a memory location, and stores the value held in a register back to the same memory location. The value initially loaded from memory is returned in the destination register.

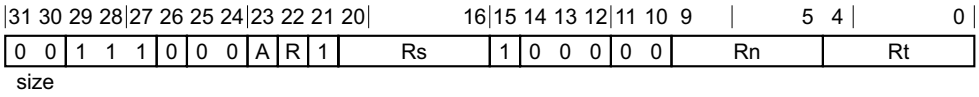
- If the destination register is not `WZR`, `SWPAB` and `SWPALB` load from memory with acquire semantics.
- `SWPLB` and `SWPALB` store to memory with release semantics.
- `SWPB` has neither acquire nor release semantics.

For more information about memory ordering semantics, see *Load-Acquire, Load-AcquirePC, and Store-Release*.

For information about memory accesses, see *Load/store addressing modes*.

Integer

(FEAT_LSE)



SWPAB variant

Applies when `A == 1 && R == 0`.

```
SWPAB <Ws>, <Wt>, [<Xn|SP>]
```

SWPALB variant

Applies when `A == 1 && R == 1`.

```
SWPALB <Ws>, <Wt>, [<Xn|SP>]
```

SWPB variant

Applies when `A == 0 && R == 0`.

```
SWPB <Ws>, <Wt>, [<Xn|SP>]
```

SWPLB variant

Applies when `A == 0 && R == 1`.

```
SWPLB <Ws>, <Wt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LSE) then UNDEFINED;

integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

boolean acquire = A == '1' && Rt != '11111';
boolean release = R == '1';
boolean tagchecked = n != 31;
```

Assembler symbols

`<Ws>` Is the 32-bit name of the general-purpose register to be stored, encoded in the "Rs" field.

<Wt> Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(8) data;
bits(8) store_value;
AccessDescriptor accdesc = CreateAccDescAtomicOp(MemAtomicOp_SWP, acquire, release, tagchecked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

store_value = X[s, 8];

bits(8) comparevalue = bits(8) UNKNOWN;    // Irrelevant when not executing CAS
data = MemAtomic(address, comparevalue, store_value, accdesc);

X[t, 32] = ZeroExtend(data, 32);
```

C6.2.411 SWPH, SWPAH, SWPALH, SWPLH

Swap halfword in memory atomically loads a 16-bit halfword from a memory location, and stores the value held in a register back to the same memory location. The value initially loaded from memory is returned in the destination register.

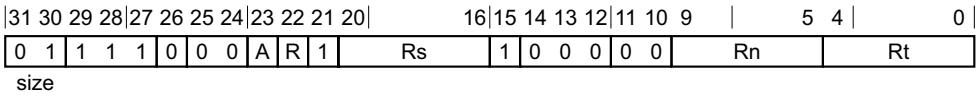
- If the destination register is not WZR, SWPAH and SWPALH load from memory with acquire semantics.
- SWPLH and SWPALH store to memory with release semantics.
- SWPH has neither acquire nor release semantics.

For more information about memory ordering semantics, see *Load-Acquire, Load-AcquirePC, and Store-Release*.

For information about memory accesses, see *Load/store addressing modes*.

Integer

(FEAT_LSE)



SWPAH variant

Applies when A == 1 && R == 0.

```
SWPAH <Ws>, <Wt>, [<Xn|SP>]
```

SWPALH variant

Applies when A == 1 && R == 1.

```
SWPALH <Ws>, <Wt>, [<Xn|SP>]
```

SWPH variant

Applies when A == 0 && R == 0.

```
SWPH <Ws>, <Wt>, [<Xn|SP>]
```

SWPLH variant

Applies when A == 0 && R == 1.

```
SWPLH <Ws>, <Wt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LSE) then UNDEFINED;

integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

boolean acquire = A == '1' && Rt != '11111';
boolean release = R == '1';
boolean tagchecked = n != 31;
```

Assembler symbols

<Ws> Is the 32-bit name of the general-purpose register to be stored, encoded in the "Rs" field.

<Wt> Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(16) data;
bits(16) store_value;
AccessDescriptor accdesc = CreateAccDescAtomicOp(MemAtomicOp_SWP, acquire, release, tagchecked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

store_value = X[s, 16];

bits(16) comparevalue = bits(16) UNKNOWN;    // Irrelevant when not executing CAS
data = MemAtomic(address, comparevalue, store_value, accdesc);

X[t, 32] = ZeroExtend(data, 32);
```

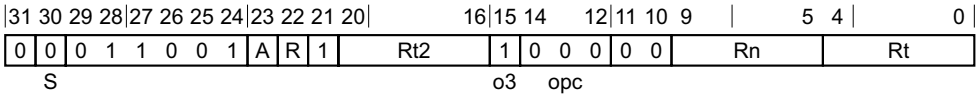
C6.2.412 SWPP, SWPPA, SWPPAL, SWPPL

Swap quadword in memory atomically loads a 128-bit quadword from a memory location, and stores the value held in a pair of registers back to the same memory location. The value initially loaded from memory is returned in the same pair of registers.

- SWPPA and SWPPAL load from memory with acquire semantics.
- SWPPL and SWPPAL store to memory with release semantics.
- SWPP has neither acquire nor release semantics.

Integer

(FEAT_LSE128)



SWPP variant

Applies when A == 0 && R == 0.

```
SWPP <Xt1>, <Xt2>, [<Xn|SP>]
```

SWPPA variant

Applies when A == 1 && R == 0.

```
SWPPA <Xt1>, <Xt2>, [<Xn|SP>]
```

SWPPAL variant

Applies when A == 1 && R == 1.

```
SWPPAL <Xt1>, <Xt2>, [<Xn|SP>]
```

SWPPL variant

Applies when A == 0 && R == 1.

```
SWPPL <Xt1>, <Xt2>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_LSE128) then UNDEFINED;
if Rt == '11111' then UNDEFINED;
if Rt2 == '11111' then UNDEFINED;
integer t = UInt(Rt);
integer t2 = UInt(Rt2);
integer n = UInt(Rn);

boolean acquire = A == '1';
boolean release = R == '1';
boolean tagchecked = n != 31;

boolean rt_unknown = FALSE;

if t == t2 then
    Constraint c = ConstrainUnpredictable(Unpredictable_LSE128OVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN rt_unknown = TRUE;    // result is UNKNOWN
```

```
        when Constraint_UNDEF   UNDEFINED;
        when Constraint_NOP     EndOfInstruction();
```

Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE Behaviors*, and particularly *CONSTRAINED UNPREDICTABLE behavior for A64 instructions*.

Assembler symbols

<Xt1>	Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.
<Xt2>	Is the 64-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(64) value1 = X[t, 64];
bits(64) value2 = X[t2, 64];
bits(128) data;
bits(128) store_value;

AccessDescriptor accdesc = CreateAccDescAtomicOp(MemAtomicOp_SWP, acquire, release, tagchecked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

store_value = if BigEndian(accdesc.acctype) then value1:value2 else value2:value1;

bits(128) comparevalue = bits(128) UNKNOWN;    // Irrelevant when not executing CAS
data = MemAtomic(address, comparevalue, store_value, accdesc);

if rt_unknown then
    data = bits(128) UNKNOWN;

if BigEndian(accdesc.acctype) then
    X[t, 64] = data<127:64>;
    X[t2, 64] = data<63:0>;
else
    X[t, 64] = data<63:0>;
    X[t2, 64] = data<127:64>;
```

Operational information

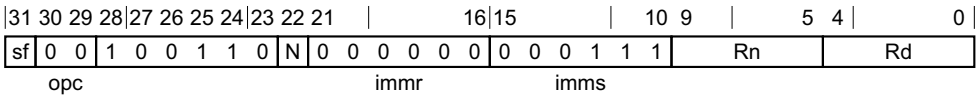
If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

C6.2.413 SXTB

Signed Extend Byte extracts an 8-bit value from a register, sign-extends it to the size of the register, and writes the result to the destination register.

This instruction is an alias of the SBFM instruction. This means that:

- The encodings in this description are named to match the encodings of SBFM.
- The description of SBFM gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.



32-bit variant

Applies when `sf == 0 && N == 0`.

```
SXTB <Wd>, <Wn>
```

is equivalent to

```
SBFM <Wd>, <Wn>, #0, #7
```

and is always the preferred disassembly.

64-bit variant

Applies when `sf == 1 && N == 1`.

```
SXTB <Xd>, <Wn>
```

is equivalent to

```
SBFM <Xd>, <Xn>, #0, #7
```

and is always the preferred disassembly.

Assembler symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
<Wn>	Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.

Operation

The description of SBFM gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.

  - The values of the NZCV flags.

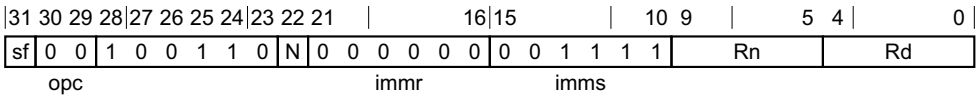
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

C6.2.414 SXTH

Sign Extend Halfword extracts a 16-bit value, sign-extends it to the size of the register, and writes the result to the destination register.

This instruction is an alias of the SBFM instruction. This means that:

- The encodings in this description are named to match the encodings of SBFM.
- The description of SBFM gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.



32-bit variant

Applies when `sf == 0 && N == 0`.

`SXTH <Wd>, <Wn>`

is equivalent to

`SBFM <Wd>, <Wn>, #0, #15`

and is always the preferred disassembly.

64-bit variant

Applies when `sf == 1 && N == 1`.

`SXTH <Xd>, <Wn>`

is equivalent to

`SBFM <Xd>, <Xn>, #0, #15`

and is always the preferred disassembly.

Assembler symbols

`<Wd>`	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
`<Xd>`	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
`<Xn>`	Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
`<Wn>`	Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.

Operation

The description of SBFM gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.

  - The values of the NZCV flags.

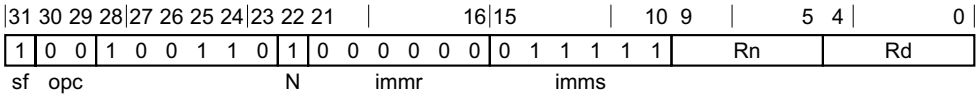
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

C6.2.415 SXTW

Sign Extend Word sign-extends a word to the size of the register, and writes the result to the destination register.

This instruction is an alias of the SBFM instruction. This means that:

- The encodings in this description are named to match the encodings of SBFM.
- The description of SBFM gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.



64-bit variant

```
SXTW <Xd>, <Wn>
```

is equivalent to

```
SBFM <Xd>, <Xn>, #0, #31
```

and is always the preferred disassembly.

Assembler symbols

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Xn> Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.

<Wn> Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.

Operation

The description of SBFM gives the operational pseudocode for this instruction.

Operational information

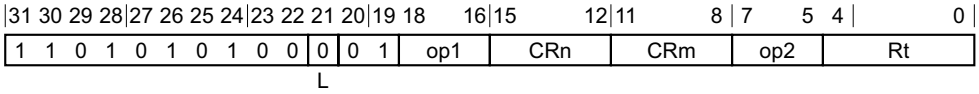
If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

C6.2.416 SYS

System instruction. For more information, see *op0==0b01, cache maintenance, TLB maintenance, address translation, prediction restriction, BRBE, Trace Extension, and Guarded Control Stack instructions* for the encodings of System instructions.

This instruction is used by the aliases AT, BRB, CFP, COSP, CPP, DC, DVP, GCSPOPCX, GCSPOPX, GCSPUSHM, GCSPUSHX, GCSSS1, IC, TLBI, and TRCIT. See *Alias conditions* for details of when each alias is preferred.



Encoding

```
SYS #<op1>, <Cn>, <Cm>, #<op2>{, <Xt>}
```

Decode for this encoding

```
AArch64.CheckSystemAccess('01', op1, CRn, CRm, op2, Rt, L);

integer t = UInt(Rt);

integer sys_op1 = UInt(op1);
integer sys_op2 = UInt(op2);
integer sys_crn = UInt(CRn);
integer sys_crm = UInt(CRm);
```

Alias conditions

Alias	is preferred when
AT	CRn == '0111' && CRm == '100x' && SysOp(op1,'0111',CRm,op2) == Sys_AT
BRB	op1 == '001' && CRn == '0111' && CRm == '0010' && SysOp('001','0111','0010',op2) == Sys_BRB
CFP	op1 == '011' && CRn == '0111' && CRm == '0011' && op2 == '100'
COSP	op1 == '011' && CRn == '0111' && CRm == '0011' && op2 == '110'
CPP	op1 == '011' && CRn == '0111' && CRm == '0011' && op2 == '111'
DC	CRn == '0111' && SysOp(op1,'0111',CRm,op2) == Sys_DC
DVP	op1 == '011' && CRn == '0111' && CRm == '0011' && op2 == '101'
GCSPOPCX	op1 == '000' && CRn == '0111' && CRm == '0111' && op2 == '101'
GCSPOPX	op1 == '000' && CRn == '0111' && CRm == '0111' && op2 == '110'
GCSPUSHM	op1 == '011' && CRn == '0111' && CRm == '0111' && op2 == '000'
GCSPUSHX	op1 == '000' && CRn == '0111' && CRm == '0111' && op2 == '100'
GCSSS1	op1 == '011' && CRn == '0111' && CRm == '0111' && op2 == '010'
IC	CRn == '0111' && SysOp(op1,'0111',CRm,op2) == Sys_IC
TLBI	CRn == '100x' && SysOp(op1,CRn,CRm,op2) == Sys_TLBI
TRCIT	op1 == '011' && CRn == '0111' && CRm == '0010' && op2 == '111'

Assembler symbols

<op1>	Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op1" field.
<Cn>	Is a name 'Cn', with 'n' in the range 0 to 15, encoded in the "CRn" field.
<Cm>	Is a name 'Cm', with 'm' in the range 0 to 15, encoded in the "CRm" field.
<op2>	Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op2" field.
<Xt>	Is the 64-bit name of the optional general-purpose source register, defaulting to '11111', encoded in the "Rt" field.

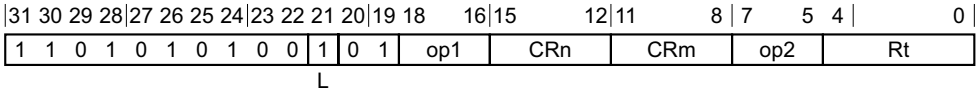
Operation

```
AArch64.SysInstr(1, sys_op1, sys_crn, sys_crm, sys_op2, t);
```

C6.2.417 SYSL

System instruction with result. For more information, see *op0==0b01, cache maintenance, TLB maintenance, address translation, prediction restriction, BRBE, Trace Extension, and Guarded Control Stack instructions* for the encodings of System instructions.

This instruction is used by the aliases GCSPOPM and GCSSS2. See *Alias conditions* for details of when each alias is preferred.



Encoding

```
SYSL <Xt>, #<op1>, <Cn>, <Cm>, #<op2>
```

Decode for this encoding

```
AArch64.CheckSystemAccess('01', op1, CRn, CRm, op2, Rt, L);

integer t = UInt(Rt);

integer sys_op1 = UInt(op1);
integer sys_op2 = UInt(op2);
integer sys_crn = UInt(CRn);
integer sys_crm = UInt(CRm);
```

Alias conditions

Alias	is preferred when
GCSPOPM	`op1 == '011' && CRn == '0111' && CRm == '0111' && op2 == '001'`
GCSSS2	`op1 == '011' && CRn == '0111' && CRm == '0111' && op2 == '011'`

Assembler symbols

<Xt>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rt" field.
<op1>	Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op1" field.
<Cn>	Is a name 'Cn', with 'n' in the range 0 to 15, encoded in the "CRn" field.
<Cm>	Is a name 'Cm', with 'm' in the range 0 to 15, encoded in the "CRm" field.
<op2>	Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op2" field.

Operation

```
// No architecturally defined instructions here.
AArch64.SysInstrWithResult(1, sys_op1, sys_crn, sys_crm, sys_op2, t);
```

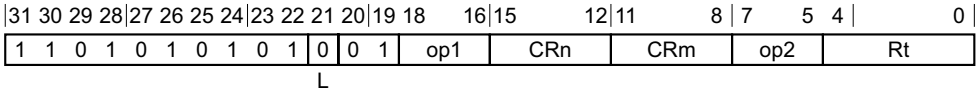
C6.2.418 SYSP

128-bit System instruction.

This instruction is used by the alias TLBIP. See *Alias conditions* for details of when each alias is preferred.

System

(FEAT_SYSINSTR128)



Encoding

```
SYSP #<op1>, <Cn>, <Cm>, #<op2>{, <Xt1>, <Xt2>}
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SYSINSTR128) then UNDEFINED;
if Rt<0> == '1' && Rt != '11111' then UNDEFINED;
AArch64.CheckSystemAccess('01', op1, CRn, CRm, op2, Rt, L);

integer t = UInt(Rt);
integer t2 = if t == 31 then 31 else UInt(Rt) + 1;

integer sys_op1 = UInt(op1);
integer sys_op2 = UInt(op2);
integer sys_crn = UInt(CRn);
integer sys_crm = UInt(CRm);
```

Alias conditions

Alias	is preferred when
TLBIP	`CRn == '100x' && SysOp(op1,CRn,CRm,op2) == Sys_TLBIP`

Assembler symbols

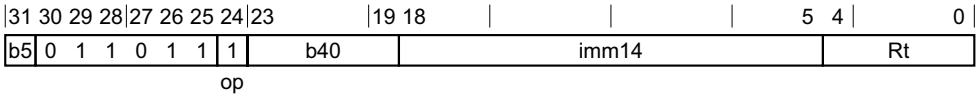
<op1>	Is a 3-bit unsigned immediate, in the range 0 to 6, encoded in the "op1" field.
<Cn>	Is a name 'Cn', with 'n' in the range 8 to 9, encoded in the "CRn" field.
<Cm>	Is a name 'Cm', with 'm' in the range 0 to 7, encoded in the "CRm" field.
<op2>	Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op2" field.
<Xt1>	Is the 64-bit name of the first optional general-purpose source register, defaulting to '11111', encoded in the "Rt" field.
<Xt2>	Is the 64-bit name of the second optional general-purpose source register, defaulting to '11111', encoded as "Rt" +1. Defaults to '11111' if "Rt" = '11111'.

Operation

```
AArch64.SysInstr128(1, sys_op1, sys_crn, sys_crm, sys_op2, t, t2);
```

C6.2.419 TBNZ

Test bit and Branch if Nonzero compares the value of a bit in a general-purpose register with zero, and conditionally branches to a label at a PC-relative offset if the comparison is not equal. It provides a hint that this is not a subroutine call or return. This instruction does not affect condition flags.



Encoding

```
TBNZ <R><t>, #<imm>, <label>
```

Decode for this encoding

```
integer t = UInt(Rt);

constant integer datasize = 32 << UInt(b5);
integer bit_pos = UInt(b5:b40);
bits(64) offset = SignExtend(imm14:'00', 64);
```

Assembler symbols

<R> Is a width specifier, encoded in the "b5" field. It can have the following values:

W when b5 = 0

X when b5 = 1

In assembler source code an 'X' specifier is always permitted, but a 'W' specifier is only permitted when the bit number is less than 32.

<t> Is the number [0-30] of the general-purpose register to be tested or the name ZR (31), encoded in the "Rt" field.

<imm> Is the bit number to be tested, in the range 0 to 63, encoded in "b5:b40".

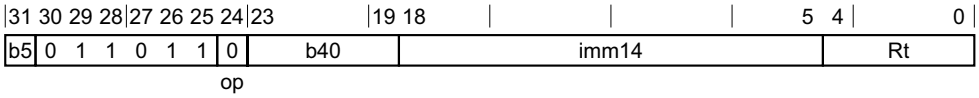
<label> Is the program label to be conditionally branched to. Its offset from the address of this instruction, in the range +/-32KB, is encoded as "imm14" times 4.

Operation

```
bits(datasize) operand = X[t, datasize];
if operand<bit_pos> == op then
    BranchTo(PC64 + offset, BranchType_DIR, TRUE);
else
    BranchNotTaken(BranchType_DIR, TRUE);
```

C6.2.420 TBZ

Test bit and Branch if Zero compares the value of a test bit with zero, and conditionally branches to a label at a PC-relative offset if the comparison is equal. It provides a hint that this is not a subroutine call or return. This instruction does not affect condition flags.



Encoding

```
TBZ <R><t>, #<imm>, <label>
```

Decode for this encoding

```
integer t = UInt(Rt);

constant integer datasize = 32 << UInt(b5);
integer bit_pos = UInt(b5:b40);
bits(64) offset = SignExtend(imm14:'00', 64);
```

Assembler symbols

<R> Is a width specifier, encoded in the "b5" field. It can have the following values:

W when b5 = 0

X when b5 = 1

In assembler source code an 'X' specifier is always permitted, but a 'W' specifier is only permitted when the bit number is less than 32.

<t> Is the number [0-30] of the general-purpose register to be tested or the name ZR (31), encoded in the "Rt" field.

<imm> Is the bit number to be tested, in the range 0 to 63, encoded in "b5:b40".

<label> Is the program label to be conditionally branched to. Its offset from the address of this instruction, in the range +/-32KB, is encoded as "imm14" times 4.

Operation

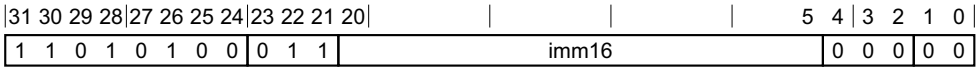
```
bits(datasize) operand = X[t, datasize];
if operand<bit_pos> == op then
    BranchTo(PC64 + offset, BranchType_DIR, TRUE);
else
    BranchNotTaken(BranchType_DIR, TRUE);
```

C6.2.421 TCANCEL

This instruction exits Transactional state and discards all state modifications that were performed transactionally. Execution continues at the instruction that follows the TSTART instruction of the outer transaction. The destination register of the TSTART instruction of the outer transaction is written with the immediate operand of TCANCEL.

System

(FEAT_TME)



Encoding

```
TCANCEL #<imm>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_TME) then UNDEFINED;
boolean retry = (imm16<15> == '1');
bits(15) reason = imm16<14:0>;
```

Assembler symbols

<imm> Is a 16-bit unsigned immediate, in the range 0 to 65535, encoded in the "imm16" field.

Operation

```
if !IsTMEEnabled() then UNDEFINED;

if TSTATE.depth > 0 then
    FailTransaction(TMFailure_CNCL, retry, FALSE, reason);
```

C6.2.422 TCOMMIT

This instruction commits the current transaction. If the current transaction is an outer transaction, then Transactional state is exited, and all state modifications performed transactionally are committed to the architectural state. TCOMMIT takes no inputs and returns no value.

Execution of TCOMMIT is UNDEFINED in Non-transactional state.

System

(FEAT_TME)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	1	0	1	0	0	0	0	0	0	1	1	0	0	1	1	0	0	0	0	0	1	1	1	1	1	1	1

Encoding

TCOMMIT

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_TME) then UNDEFINED;
```

Operation

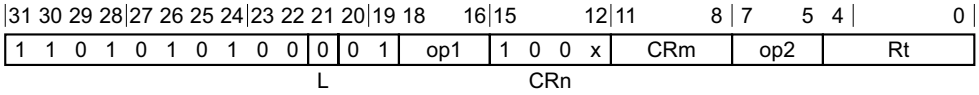
```
if !IsTMEEnabled() then UNDEFINED;

if TSTATE.depth == 0 then
    UNDEFINED;

if TSTATE.depth == 1 then
    CommitTransactionalWrites();
    ClearExclusiveLocal(ProcessorID());

TSTATE.depth = TSTATE.depth - 1;
```

C6.2.423 TLBI

TLB Invalidate operation. For more information, see *op0==0b01, cache maintenance, TLB maintenance, address translation, prediction restriction, BRBE, Trace Extension, and Guarded Control Stack instructions*.

This instruction is an alias of the SYS instruction. This means that:

- The encodings in this description are named to match the encodings of SYS.
- The description of SYS gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.



Encoding

```
TLBI <tlbi_op>{, <Xt>}
```

is equivalent to

```
SYS #<op1>, <Cn>, <Cm>, #<op2>{, <Xt>}
```

and is the preferred disassembly when `SysOp(op1,CRn,CRm,op2) == Sys_TLBI`.

Assembler symbols

`<op1>` Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op1" field.

`<Cn>` Is a name 'Cn', with 'n' in the range 0 to 15, encoded in the "CRn" field.

`<Cm>` Is a name 'Cm', with 'm' in the range 0 to 15, encoded in the "CRm" field.

`<op2>` Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op2" field.

`<tlbi_op>` Is a TLBI instruction name, as listed for the TLBI system instruction group, encoded in the "op1:CRn:CRm:op2" field. It can have the following values:

`VMALLE1IS`	when `op1 = 000, CRn = 1000, CRm = 0011, op2 = 000`
`VAE1IS`	when `op1 = 000, CRn = 1000, CRm = 0011, op2 = 001`
`ASIDE1IS`	when `op1 = 000, CRn = 1000, CRm = 0011, op2 = 010`
`VAAE1IS`	when `op1 = 000, CRn = 1000, CRm = 0011, op2 = 011`
`VALE1IS`	when `op1 = 000, CRn = 1000, CRm = 0011, op2 = 101`
`VAALE1IS`	when `op1 = 000, CRn = 1000, CRm = 0011, op2 = 111`
`VMALLE1`	when `op1 = 000, CRn = 1000, CRm = 0111, op2 = 000`
`VAE1`	when `op1 = 000, CRn = 1000, CRm = 0111, op2 = 001`
`ASIDE1`	when `op1 = 000, CRn = 1000, CRm = 0111, op2 = 010`
`VAAE1`	when `op1 = 000, CRn = 1000, CRm = 0111, op2 = 011`
`VALE1`	when `op1 = 000, CRn = 1000, CRm = 0111, op2 = 101`
`VAALE1`	when `op1 = 000, CRn = 1000, CRm = 0111, op2 = 111`
`IPAS2E1IS`	when `op1 = 100, CRn = 1000, CRm = 0000, op2 = 001`
`IPAS2LE1IS`	when `op1 = 100, CRn = 1000, CRm = 0000, op2 = 101`
`ALLE2IS`	when `op1 = 100, CRn = 1000, CRm = 0011, op2 = 000`

VAE2IS	when op1 = 100, CRn = 1000, CRm = 0011, op2 = 001
ALLE1IS	when op1 = 100, CRn = 1000, CRm = 0011, op2 = 100
VALE2IS	when op1 = 100, CRn = 1000, CRm = 0011, op2 = 101
VMALLS12E1IS	when op1 = 100, CRn = 1000, CRm = 0011, op2 = 110
IPAS2E1	when op1 = 100, CRn = 1000, CRm = 0100, op2 = 001
IPAS2LE1	when op1 = 100, CRn = 1000, CRm = 0100, op2 = 101
ALLE2	when op1 = 100, CRn = 1000, CRm = 0111, op2 = 000
VAE2	when op1 = 100, CRn = 1000, CRm = 0111, op2 = 001
ALLE1	when op1 = 100, CRn = 1000, CRm = 0111, op2 = 100
VALE2	when op1 = 100, CRn = 1000, CRm = 0111, op2 = 101
VMALLS12E1	when op1 = 100, CRn = 1000, CRm = 0111, op2 = 110
ALLE3IS	when op1 = 110, CRn = 1000, CRm = 0011, op2 = 000
VAE3IS	when op1 = 110, CRn = 1000, CRm = 0011, op2 = 001
VALE3IS	when op1 = 110, CRn = 1000, CRm = 0011, op2 = 101
ALLE3	when op1 = 110, CRn = 1000, CRm = 0111, op2 = 000
VAE3	when op1 = 110, CRn = 1000, CRm = 0111, op2 = 001
VALE3	when op1 = 110, CRn = 1000, CRm = 0111, op2 = 101

When FEAT_TLBIOS is implemented, the following values are also valid:

VMALLE1OS	when op1 = 000, CRn = 1000, CRm = 0001, op2 = 000
VAE1OS	when op1 = 000, CRn = 1000, CRm = 0001, op2 = 001
ASIDE1OS	when op1 = 000, CRn = 1000, CRm = 0001, op2 = 010
VAAE1OS	when op1 = 000, CRn = 1000, CRm = 0001, op2 = 011
VALE1OS	when op1 = 000, CRn = 1000, CRm = 0001, op2 = 101
VAALE1OS	when op1 = 000, CRn = 1000, CRm = 0001, op2 = 111
ALLE2OS	when op1 = 100, CRn = 1000, CRm = 0001, op2 = 000
VAE2OS	when op1 = 100, CRn = 1000, CRm = 0001, op2 = 001
ALLE1OS	when op1 = 100, CRn = 1000, CRm = 0001, op2 = 100
VALE2OS	when op1 = 100, CRn = 1000, CRm = 0001, op2 = 101
VMALLS12E1OS	when op1 = 100, CRn = 1000, CRm = 0001, op2 = 110
IPAS2E1OS	when op1 = 100, CRn = 1000, CRm = 0100, op2 = 000
IPAS2LE1OS	when op1 = 100, CRn = 1000, CRm = 0100, op2 = 100
ALLE3OS	when op1 = 110, CRn = 1000, CRm = 0001, op2 = 000
VAE3OS	when op1 = 110, CRn = 1000, CRm = 0001, op2 = 001
VALE3OS	when op1 = 110, CRn = 1000, CRm = 0001, op2 = 101

When FEAT_TLBIRANGE is implemented, the following values are also valid:

RVAE1IS	when op1 = 000, CRn = 1000, CRm = 0010, op2 = 001
RVAAE1IS	when op1 = 000, CRn = 1000, CRm = 0010, op2 = 011
RVALE1IS	when op1 = 000, CRn = 1000, CRm = 0010, op2 = 101
RVAALE1IS	when op1 = 000, CRn = 1000, CRm = 0010, op2 = 111
RVAE1OS	when op1 = 000, CRn = 1000, CRm = 0101, op2 = 001
RVAAE1OS	when op1 = 000, CRn = 1000, CRm = 0101, op2 = 011
RVALE1OS	when op1 = 000, CRn = 1000, CRm = 0101, op2 = 101
RVAALE1OS	when op1 = 000, CRn = 1000, CRm = 0101, op2 = 111
RVAE1	when op1 = 000, CRn = 1000, CRm = 0110, op2 = 001

RVAAE1 when op1 = 000, CRn = 1000, CRm = 0110, op2 = 011
RVALE1 when op1 = 000, CRn = 1000, CRm = 0110, op2 = 101
RVAALE1 when op1 = 000, CRn = 1000, CRm = 0110, op2 = 111
RIPAS2E1IS when op1 = 100, CRn = 1000, CRm = 0000, op2 = 010
RIPAS2LE1IS when op1 = 100, CRn = 1000, CRm = 0000, op2 = 110
RVAE2IS when op1 = 100, CRn = 1000, CRm = 0010, op2 = 001
RVALE2IS when op1 = 100, CRn = 1000, CRm = 0010, op2 = 101
RIPAS2E1 when op1 = 100, CRn = 1000, CRm = 0100, op2 = 010
RIPAS2E1OS when op1 = 100, CRn = 1000, CRm = 0100, op2 = 011
RIPAS2LE1 when op1 = 100, CRn = 1000, CRm = 0100, op2 = 110
RIPAS2LE1OS when op1 = 100, CRn = 1000, CRm = 0100, op2 = 111
RVAE2OS when op1 = 100, CRn = 1000, CRm = 0101, op2 = 001
RVALE2OS when op1 = 100, CRn = 1000, CRm = 0101, op2 = 101
RVAE2 when op1 = 100, CRn = 1000, CRm = 0110, op2 = 001
RVALE2 when op1 = 100, CRn = 1000, CRm = 0110, op2 = 101
RVAE3IS when op1 = 110, CRn = 1000, CRm = 0010, op2 = 001
RVALE3IS when op1 = 110, CRn = 1000, CRm = 0010, op2 = 101
RVAE3OS when op1 = 110, CRn = 1000, CRm = 0101, op2 = 001
RVALE3OS when op1 = 110, CRn = 1000, CRm = 0101, op2 = 101
RVAE3 when op1 = 110, CRn = 1000, CRm = 0110, op2 = 001
RVALE3 when op1 = 110, CRn = 1000, CRm = 0110, op2 = 101

When FEAT_XS is implemented, the following values are also valid:

VMALLE1OSNXS when op1 = 000, CRn = 1001, CRm = 0001, op2 = 000
VAE1OSNXS when op1 = 000, CRn = 1001, CRm = 0001, op2 = 001
ASIDE1OSNXS when op1 = 000, CRn = 1001, CRm = 0001, op2 = 010
VAAE1OSNXS when op1 = 000, CRn = 1001, CRm = 0001, op2 = 011
VALE1OSNXS when op1 = 000, CRn = 1001, CRm = 0001, op2 = 101
VAALE1OSNXS when op1 = 000, CRn = 1001, CRm = 0001, op2 = 111
RVAE1ISNXS when op1 = 000, CRn = 1001, CRm = 0010, op2 = 001
RVAAE1ISNXS when op1 = 000, CRn = 1001, CRm = 0010, op2 = 011
RVALE1ISNXS when op1 = 000, CRn = 1001, CRm = 0010, op2 = 101
RVAALE1ISNXS when op1 = 000, CRn = 1001, CRm = 0010, op2 = 111
VMALLE1ISNXS when op1 = 000, CRn = 1001, CRm = 0011, op2 = 000
VAE1ISNXS when op1 = 000, CRn = 1001, CRm = 0011, op2 = 001
ASIDE1ISNXS when op1 = 000, CRn = 1001, CRm = 0011, op2 = 010
VAAE1ISNXS when op1 = 000, CRn = 1001, CRm = 0011, op2 = 011
VALE1ISNXS when op1 = 000, CRn = 1001, CRm = 0011, op2 = 101
VAALE1ISNXS when op1 = 000, CRn = 1001, CRm = 0011, op2 = 111
RVAE1OSNXS when op1 = 000, CRn = 1001, CRm = 0101, op2 = 001
RVAAE1OSNXS when op1 = 000, CRn = 1001, CRm = 0101, op2 = 011
RVALE1OSNXS when op1 = 000, CRn = 1001, CRm = 0101, op2 = 101
RVAALE1OSNXS when op1 = 000, CRn = 1001, CRm = 0101, op2 = 111
RVAE1NXS when op1 = 000, CRn = 1001, CRm = 0110, op2 = 001
RVAAE1NXS when op1 = 000, CRn = 1001, CRm = 0110, op2 = 011

RVALE1NXS when op1 = 000, CRn = 1001, CRm = 0110, op2 = 101

RVAALE1NXS when op1 = 000, CRn = 1001, CRm = 0110, op2 = 111

VMALLE1NXS when op1 = 000, CRn = 1001, CRm = 0111, op2 = 000

VAE1NXS when op1 = 000, CRn = 1001, CRm = 0111, op2 = 001

ASIDE1NXS when op1 = 000, CRn = 1001, CRm = 0111, op2 = 010

VAAE1NXS when op1 = 000, CRn = 1001, CRm = 0111, op2 = 011

VALE1NXS when op1 = 000, CRn = 1001, CRm = 0111, op2 = 101

VAALE1NXS when op1 = 000, CRn = 1001, CRm = 0111, op2 = 111

IPAS2E1ISNXS when op1 = 100, CRn = 1001, CRm = 0000, op2 = 001

RIPAS2E1ISNXS when op1 = 100, CRn = 1001, CRm = 0000, op2 = 010

IPAS2LE1ISNXS when op1 = 100, CRn = 1001, CRm = 0000, op2 = 101

RIPAS2LE1ISNXS when op1 = 100, CRn = 1001, CRm = 0000, op2 = 110

ALLE2OSNXS when op1 = 100, CRn = 1001, CRm = 0001, op2 = 000

VAE2OSNXS when op1 = 100, CRn = 1001, CRm = 0001, op2 = 001

ALLE1OSNXS when op1 = 100, CRn = 1001, CRm = 0001, op2 = 100

VALE2OSNXS when op1 = 100, CRn = 1001, CRm = 0001, op2 = 101

VMALLS12E1OSNXS when op1 = 100, CRn = 1001, CRm = 0001, op2 = 110

RVAE2ISNXS when op1 = 100, CRn = 1001, CRm = 0010, op2 = 001

RVALE2ISNXS when op1 = 100, CRn = 1001, CRm = 0010, op2 = 101

ALLE2ISNXS when op1 = 100, CRn = 1001, CRm = 0011, op2 = 000

VAE2ISNXS when op1 = 100, CRn = 1001, CRm = 0011, op2 = 001

ALLE1ISNXS when op1 = 100, CRn = 1001, CRm = 0011, op2 = 100

VALE2ISNXS when op1 = 100, CRn = 1001, CRm = 0011, op2 = 101

VMALLS12E1ISNXS when op1 = 100, CRn = 1001, CRm = 0011, op2 = 110

IPAS2E1OSNXS when op1 = 100, CRn = 1001, CRm = 0100, op2 = 000

IPAS2E1NXS when op1 = 100, CRn = 1001, CRm = 0100, op2 = 001

RIPAS2E1NXS when op1 = 100, CRn = 1001, CRm = 0100, op2 = 010

RIPAS2E1OSNXS when op1 = 100, CRn = 1001, CRm = 0100, op2 = 011

IPAS2LE1OSNXS when op1 = 100, CRn = 1001, CRm = 0100, op2 = 100

IPAS2LE1NXS when op1 = 100, CRn = 1001, CRm = 0100, op2 = 101

RIPAS2LE1NXS when op1 = 100, CRn = 1001, CRm = 0100, op2 = 110

RIPAS2LE1OSNXS when op1 = 100, CRn = 1001, CRm = 0100, op2 = 111

RVAE2OSNXS when op1 = 100, CRn = 1001, CRm = 0101, op2 = 001

RVALE2OSNXS when op1 = 100, CRn = 1001, CRm = 0101, op2 = 101

RVAE2NXS when op1 = 100, CRn = 1001, CRm = 0110, op2 = 001

RVALE2NXS when op1 = 100, CRn = 1001, CRm = 0110, op2 = 101

ALLE2NXS when op1 = 100, CRn = 1001, CRm = 0111, op2 = 000

VAE2NXS when op1 = 100, CRn = 1001, CRm = 0111, op2 = 001

ALLE1NXS when op1 = 100, CRn = 1001, CRm = 0111, op2 = 100

VALE2NXS when op1 = 100, CRn = 1001, CRm = 0111, op2 = 101

VMALLS12E1NXS when op1 = 100, CRn = 1001, CRm = 0111, op2 = 110

ALLE3OSNXS when op1 = 110, CRn = 1001, CRm = 0001, op2 = 000

VAE3OSNXS when op1 = 110, CRn = 1001, CRm = 0001, op2 = 001

VALE3OSNXS when op1 = 110, CRn = 1001, CRm = 0001, op2 = 101

RVAE3ISNXS when op1 = 110, CRn = 1001, CRm = 0010, op2 = 001

RVALE3ISNXS when op1 = 110, CRn = 1001, CRm = 0010, op2 = 101

ALLE3ISNXS when op1 = 110, CRn = 1001, CRm = 0011, op2 = 000

VAE3ISNXS when op1 = 110, CRn = 1001, CRm = 0011, op2 = 001

VALE3ISNXS when op1 = 110, CRn = 1001, CRm = 0011, op2 = 101

RVAE3OSNXS when op1 = 110, CRn = 1001, CRm = 0101, op2 = 001

RVALE3OSNXS when op1 = 110, CRn = 1001, CRm = 0101, op2 = 101

RVAE3NXS when op1 = 110, CRn = 1001, CRm = 0110, op2 = 001

RVALE3NXS when op1 = 110, CRn = 1001, CRm = 0110, op2 = 101

ALLE3NXS when op1 = 110, CRn = 1001, CRm = 0111, op2 = 000

VAE3NXS when op1 = 110, CRn = 1001, CRm = 0111, op2 = 001

VALE3NXS when op1 = 110, CRn = 1001, CRm = 0111, op2 = 101

When FEAT_RME is implemented, the following values are also valid:

PAALLOS when op1 = 110, CRn = 1000, CRm = 0001, op2 = 100

RPAOS when op1 = 110, CRn = 1000, CRm = 0100, op2 = 011

RPALOS when op1 = 110, CRn = 1000, CRm = 0100, op2 = 111

PAALL when op1 = 110, CRn = 1000, CRm = 0111, op2 = 100

<Xt> Is the 64-bit name of the optional general-purpose source register, defaulting to '11111', encoded in the "Rt" field.

Operation

The description of SYS gives the operational pseudocode for this instruction.

C6.2.424 TLBIP

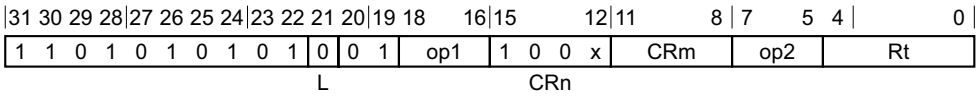
TLB Invalidate Pair operation.

This instruction is an alias of the SYSP instruction. This means that:

- The encodings in this description are named to match the encodings of SYSP.
- The description of SYSP gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

System

(FEAT_D128)



Encoding

`TLBIP <tlbip_op>{, <Xt1>, <Xt2>}`

is equivalent to

`SYSP #<op1>, <Cn>, <Cm>, #<op2>{, <Xt1>, <Xt2>}`

and is the preferred disassembly when `SysOp(op1,CRn,CRm,op2) == Sys_TLBIP`.

Assembler symbols

`<op1>` Is a 3-bit unsigned immediate, in the range 0 to 6, encoded in the "op1" field.

`<Cn>` Is a name 'Cn', with 'n' in the range 8 to 9, encoded in the "CRn" field.

`<Cm>` Is a name 'Cm', with 'm' in the range 0 to 7, encoded in the "CRm" field.

`<op2>` Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op2" field.

`<tlbip_op>` Is a TLBIP instruction name, as listed for the TLBIP system pair instruction group, encoded in the "op1:CRn:CRm:op2" field. It can have the following values:

`VAE1OS`	when `op1 = 000, CRn = 1000, CRm = 0001, op2 = 001`
`VAAE1OS`	when `op1 = 000, CRn = 1000, CRm = 0001, op2 = 011`
`VALE1OS`	when `op1 = 000, CRn = 1000, CRm = 0001, op2 = 101`
`VAALE1OS`	when `op1 = 000, CRn = 1000, CRm = 0001, op2 = 111`
`RVAE1IS`	when `op1 = 000, CRn = 1000, CRm = 0010, op2 = 001`
`RVAAE1IS`	when `op1 = 000, CRn = 1000, CRm = 0010, op2 = 011`
`RVALE1IS`	when `op1 = 000, CRn = 1000, CRm = 0010, op2 = 101`
`RVAALE1IS`	when `op1 = 000, CRn = 1000, CRm = 0010, op2 = 111`
`VAE1IS`	when `op1 = 000, CRn = 1000, CRm = 0011, op2 = 001`
`VAAE1IS`	when `op1 = 000, CRn = 1000, CRm = 0011, op2 = 011`
`VALE1IS`	when `op1 = 000, CRn = 1000, CRm = 0011, op2 = 101`
`VAALE1IS`	when `op1 = 000, CRn = 1000, CRm = 0011, op2 = 111`
`RVAE1OS`	when `op1 = 000, CRn = 1000, CRm = 0101, op2 = 001`
`RVAAE1OS`	when `op1 = 000, CRn = 1000, CRm = 0101, op2 = 011`
`RVALE1OS`	when `op1 = 000, CRn = 1000, CRm = 0101, op2 = 101`

RVAALE1OS when op1 = 000, CRn = 1000, CRm = 0101, op2 = 111
RVAE1 when op1 = 000, CRn = 1000, CRm = 0110, op2 = 001
RVAAE1 when op1 = 000, CRn = 1000, CRm = 0110, op2 = 011
RVALE1 when op1 = 000, CRn = 1000, CRm = 0110, op2 = 101
RVAALE1 when op1 = 000, CRn = 1000, CRm = 0110, op2 = 111
VAE1 when op1 = 000, CRn = 1000, CRm = 0111, op2 = 001
VAAE1 when op1 = 000, CRn = 1000, CRm = 0111, op2 = 011
VALE1 when op1 = 000, CRn = 1000, CRm = 0111, op2 = 101
VAALE1 when op1 = 000, CRn = 1000, CRm = 0111, op2 = 111
VAE1OSNXS when op1 = 000, CRn = 1001, CRm = 0001, op2 = 001
VAAE1OSNXS when op1 = 000, CRn = 1001, CRm = 0001, op2 = 011
VALE1OSNXS when op1 = 000, CRn = 1001, CRm = 0001, op2 = 101
VAALE1OSNXS when op1 = 000, CRn = 1001, CRm = 0001, op2 = 111
RVAE1ISNXS when op1 = 000, CRn = 1001, CRm = 0010, op2 = 001
RVAAE1ISNXS when op1 = 000, CRn = 1001, CRm = 0010, op2 = 011
RVALE1ISNXS when op1 = 000, CRn = 1001, CRm = 0010, op2 = 101
RVAALE1ISNXS when op1 = 000, CRn = 1001, CRm = 0010, op2 = 111
VAE1ISNXS when op1 = 000, CRn = 1001, CRm = 0011, op2 = 001
VAAE1ISNXS when op1 = 000, CRn = 1001, CRm = 0011, op2 = 011
VALE1ISNXS when op1 = 000, CRn = 1001, CRm = 0011, op2 = 101
VAALE1ISNXS when op1 = 000, CRn = 1001, CRm = 0011, op2 = 111
RVAE1OSNXS when op1 = 000, CRn = 1001, CRm = 0101, op2 = 001
RVAAE1OSNXS when op1 = 000, CRn = 1001, CRm = 0101, op2 = 011
RVALE1OSNXS when op1 = 000, CRn = 1001, CRm = 0101, op2 = 101
RVAALE1OSNXS when op1 = 000, CRn = 1001, CRm = 0101, op2 = 111
RVAE1NXS when op1 = 000, CRn = 1001, CRm = 0110, op2 = 001
RVAAE1NXS when op1 = 000, CRn = 1001, CRm = 0110, op2 = 011
RVALE1NXS when op1 = 000, CRn = 1001, CRm = 0110, op2 = 101
RVAALE1NXS when op1 = 000, CRn = 1001, CRm = 0110, op2 = 111
VAE1NXS when op1 = 000, CRn = 1001, CRm = 0111, op2 = 001
VAAE1NXS when op1 = 000, CRn = 1001, CRm = 0111, op2 = 011
VALE1NXS when op1 = 000, CRn = 1001, CRm = 0111, op2 = 101
VAALE1NXS when op1 = 000, CRn = 1001, CRm = 0111, op2 = 111
IPAS2E1IS when op1 = 100, CRn = 1000, CRm = 0000, op2 = 001
RIPAS2E1IS when op1 = 100, CRn = 1000, CRm = 0000, op2 = 010
IPAS2LE1IS when op1 = 100, CRn = 1000, CRm = 0000, op2 = 101
RIPAS2LE1IS when op1 = 100, CRn = 1000, CRm = 0000, op2 = 110
VAE2OS when op1 = 100, CRn = 1000, CRm = 0001, op2 = 001
VALE2OS when op1 = 100, CRn = 1000, CRm = 0001, op2 = 101
RVAE2IS when op1 = 100, CRn = 1000, CRm = 0010, op2 = 001
RVALE2IS when op1 = 100, CRn = 1000, CRm = 0010, op2 = 101
VAE2IS when op1 = 100, CRn = 1000, CRm = 0011, op2 = 001
VALE2IS when op1 = 100, CRn = 1000, CRm = 0011, op2 = 101
IPAS2E1OS when op1 = 100, CRn = 1000, CRm = 0100, op2 = 000

IPAS2E1 when op1 = 100, CRn = 1000, CRm = 0100, op2 = 001

RIPAS2E1 when op1 = 100, CRn = 1000, CRm = 0100, op2 = 010

RIPAS2E1OS when op1 = 100, CRn = 1000, CRm = 0100, op2 = 011

IPAS2LE1OS when op1 = 100, CRn = 1000, CRm = 0100, op2 = 100

IPAS2LE1 when op1 = 100, CRn = 1000, CRm = 0100, op2 = 101

RIPAS2LE1 when op1 = 100, CRn = 1000, CRm = 0100, op2 = 110

RIPAS2LE1OS when op1 = 100, CRn = 1000, CRm = 0100, op2 = 111

RVAE2OS when op1 = 100, CRn = 1000, CRm = 0101, op2 = 001

RVALE2OS when op1 = 100, CRn = 1000, CRm = 0101, op2 = 101

RVAE2 when op1 = 100, CRn = 1000, CRm = 0110, op2 = 001

RVALE2 when op1 = 100, CRn = 1000, CRm = 0110, op2 = 101

VAE2 when op1 = 100, CRn = 1000, CRm = 0111, op2 = 001

VALE2 when op1 = 100, CRn = 1000, CRm = 0111, op2 = 101

IPAS2E1ISNXS when op1 = 100, CRn = 1001, CRm = 0000, op2 = 001

RIPAS2E1ISNXS when op1 = 100, CRn = 1001, CRm = 0000, op2 = 010

IPAS2LE1ISNXS when op1 = 100, CRn = 1001, CRm = 0000, op2 = 101

RIPAS2LE1ISNXS when op1 = 100, CRn = 1001, CRm = 0000, op2 = 110

VAE2OSNXS when op1 = 100, CRn = 1001, CRm = 0001, op2 = 001

VALE2OSNXS when op1 = 100, CRn = 1001, CRm = 0001, op2 = 101

RVAE2ISNXS when op1 = 100, CRn = 1001, CRm = 0010, op2 = 001

RVALE2ISNXS when op1 = 100, CRn = 1001, CRm = 0010, op2 = 101

VAE2ISNXS when op1 = 100, CRn = 1001, CRm = 0011, op2 = 001

VALE2ISNXS when op1 = 100, CRn = 1001, CRm = 0011, op2 = 101

IPAS2E1OSNXS when op1 = 100, CRn = 1001, CRm = 0100, op2 = 000

IPAS2E1NXS when op1 = 100, CRn = 1001, CRm = 0100, op2 = 001

RIPAS2E1NXS when op1 = 100, CRn = 1001, CRm = 0100, op2 = 010

RIPAS2E1OSNXS when op1 = 100, CRn = 1001, CRm = 0100, op2 = 011

IPAS2LE1OSNXS when op1 = 100, CRn = 1001, CRm = 0100, op2 = 100

IPAS2LE1NXS when op1 = 100, CRn = 1001, CRm = 0100, op2 = 101

RIPAS2LE1NXS when op1 = 100, CRn = 1001, CRm = 0100, op2 = 110

RIPAS2LE1OSNXS when op1 = 100, CRn = 1001, CRm = 0100, op2 = 111

RVAE2OSNXS when op1 = 100, CRn = 1001, CRm = 0101, op2 = 001

RVALE2OSNXS when op1 = 100, CRn = 1001, CRm = 0101, op2 = 101

RVAE2NXS when op1 = 100, CRn = 1001, CRm = 0110, op2 = 001

RVALE2NXS when op1 = 100, CRn = 1001, CRm = 0110, op2 = 101

VAE2NXS when op1 = 100, CRn = 1001, CRm = 0111, op2 = 001

VALE2NXS when op1 = 100, CRn = 1001, CRm = 0111, op2 = 101

VAE3OS when op1 = 110, CRn = 1000, CRm = 0001, op2 = 001

VALE3OS when op1 = 110, CRn = 1000, CRm = 0001, op2 = 101

RVAE3IS when op1 = 110, CRn = 1000, CRm = 0010, op2 = 001

RVALE3IS when op1 = 110, CRn = 1000, CRm = 0010, op2 = 101

VAE3IS when op1 = 110, CRn = 1000, CRm = 0011, op2 = 001

VALE3IS when op1 = 110, CRn = 1000, CRm = 0011, op2 = 101

RVAE3OS when op1 = 110, CRn = 1000, CRm = 0101, op2 = 001

RVALE3OS	when op1 = 110, CRn = 1000, CRm = 0101, op2 = 101
RVAE3	when op1 = 110, CRn = 1000, CRm = 0110, op2 = 001
RVALE3	when op1 = 110, CRn = 1000, CRm = 0110, op2 = 101
VAE3	when op1 = 110, CRn = 1000, CRm = 0111, op2 = 001
VALE3	when op1 = 110, CRn = 1000, CRm = 0111, op2 = 101
VAE3OSNXS	when op1 = 110, CRn = 1001, CRm = 0001, op2 = 001
VALE3OSNXS	when op1 = 110, CRn = 1001, CRm = 0001, op2 = 101
RVAE3ISNXS	when op1 = 110, CRn = 1001, CRm = 0010, op2 = 001
RVALE3ISNXS	when op1 = 110, CRn = 1001, CRm = 0010, op2 = 101
VAE3ISNXS	when op1 = 110, CRn = 1001, CRm = 0011, op2 = 001
VALE3ISNXS	when op1 = 110, CRn = 1001, CRm = 0011, op2 = 101
RVAE3OSNXS	when op1 = 110, CRn = 1001, CRm = 0101, op2 = 001
RVALE3OSNXS	when op1 = 110, CRn = 1001, CRm = 0101, op2 = 101
RVAE3NXS	when op1 = 110, CRn = 1001, CRm = 0110, op2 = 001
RVALE3NXS	when op1 = 110, CRn = 1001, CRm = 0110, op2 = 101
VAE3NXS	when op1 = 110, CRn = 1001, CRm = 0111, op2 = 001
VALE3NXS	when op1 = 110, CRn = 1001, CRm = 0111, op2 = 101

<Xt1> Is the 64-bit name of the first optional general-purpose source register, defaulting to '11111', encoded in the "Rt" field.

<Xt2> Is the 64-bit name of the second optional general-purpose source register, defaulting to '11111', encoded as "Rt" +1. Defaults to '11111' if "Rt" = '11111'.

Operation

The description of SYSP gives the operational pseudocode for this instruction.

C6.2.425 TRCIT

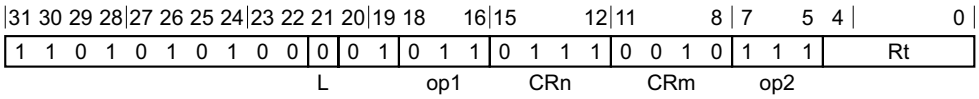
Trace Instrumentation generates an instrumentation trace packet that contains the value of the provided register.

This instruction is an alias of the SYS instruction. This means that:

- The encodings in this description are named to match the encodings of SYS.
- The description of SYS gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.

System

(FEAT_ITE)



Encoding

`TRCIT <Xt>`

is equivalent to

`SYS #3, C7, C2, #7, <Xt>`

and is always the preferred disassembly.

Assembler symbols

`<Xt>` Is the 64-bit name of the general-purpose source register, encoded in the "Rt" field.

Operation

The description of SYS gives the operational pseudocode for this instruction.

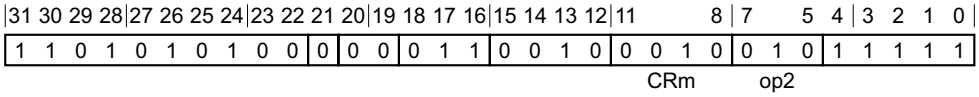
C6.2.426 TSB

Trace Synchronization Barrier. This instruction is a barrier that synchronizes the trace operations of instructions, see *Trace Synchronization Barrier (TSB)*.

If FEAT_TRF is not implemented, this instruction executes as a NOP.

System

(FEAT_TRF)



Encoding

TSB CSYNC

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_TRF) then EndOfInstruction();
```

Operation

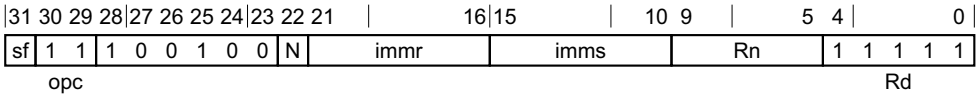
```
TraceSynchronizationBarrier();
```

C6.2.427 TST (immediate)

Test bits (immediate) , setting the condition flags and discarding the result : `Rn AND imm`

This instruction is an alias of the ANDS (immediate) instruction. This means that:

- The encodings in this description are named to match the encodings of ANDS (immediate).
- The description of ANDS (immediate) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.



32-bit variant

Applies when `sf == 0 && N == 0`.

`TST <Wn>, #<imm>`

is equivalent to

`ANDS WZR, <Wn>, #<imm>`

and is always the preferred disassembly.

64-bit variant

Applies when `sf == 1`.

`TST <Xn>, #<imm>`

is equivalent to

`ANDS XZR, <Xn>, #<imm>`

and is always the preferred disassembly.

Assembler symbols

`<Wn>`	Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
`<Xn>`	Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
`<imm>`	For the 32-bit variant: is the bitmask immediate, encoded in "imms:immr". For the 64-bit variant: is the bitmask immediate, encoded in "N:imms:immr".

Operation

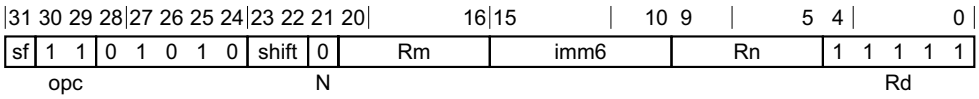
The description of ANDS (immediate) gives the operational pseudocode for this instruction.

C6.2.428 TST (shifted register)

Test (shifted register) performs a bitwise AND operation on a register value and an optionally-shifted register value. It updates the condition flags based on the result, and discards the result.

This instruction is an alias of the ANDS (shifted register) instruction. This means that:

- The encodings in this description are named to match the encodings of ANDS (shifted register).
- The description of ANDS (shifted register) gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.



32-bit variant

Applies when `sf == 0`.

```
TST <Wn>, <Wm>{, <shift> #<amount>}
```

is equivalent to

```
ANDS WZR, <Wn>, <Wm>{, <shift> #<amount>}
```

and is always the preferred disassembly.

64-bit variant

Applies when `sf == 1`.

```
TST <Xn>, <Xm>{, <shift> #<amount>}
```

is equivalent to

```
ANDS XZR, <Xn>, <Xm>{, <shift> #<amount>}
```

and is always the preferred disassembly.

Assembler symbols

`<Wn>` Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.

`<Wm>` Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.

`<Xn>` Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.

`<Xm>` Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.

`<shift>` Is the optional shift to be applied to the final source, defaulting to LSL and encoded in the "shift" field. It can have the following values:

- `LSL` when `shift = 00`
- `LSR` when `shift = 01`
- `ASR` when `shift = 10`
- `ROR` when `shift = 11`

`<amount>` For the 32-bit variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field.

For the 64-bit variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field,

Operation

The description of ANDS (shifted register) gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

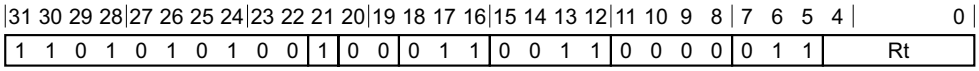
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

C6.2.429 TSTART

This instruction starts a new transaction. If the transaction started successfully, the destination register is set to zero. If the transaction failed or was canceled, then all state modifications that were performed transactionally are discarded and the destination register is written with a nonzero value that encodes the cause of the failure.

System

(FEAT_TME)



Encoding

TSTART <Xt>

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_TME) then UNDEFINED;
integer t = UInt(Rt);
```

Assembler symbols

<Xt> Is the 64-bit name of the general-purpose destination register, encoded in the "Rt" field.

Operation

```
if !IsTMEEnabled() then UNDEFINED;

boolean IsEL1Regime;
bit tme;
bit tmt;
case PSTATE.EL of
    when EL0
        IsEL1Regime = S1TranslationRegime() == EL1;
        if IsEL1Regime then
            tme = SCTLR_EL1.TME0;
            tmt = SCTLR_EL1.TMT0;
        else
            tme = SCTLR_EL2.TME0;
            tmt = SCTLR_EL2.TMT0;
    when EL1
        tme = SCTLR_EL1.TME;
        tmt = SCTLR_EL1.TMT;
    when EL2
        tme = SCTLR_EL2.TME;
        tmt = SCTLR_EL2.TMT;
    when EL3
        tme = SCTLR_EL3.TME;
        tmt = SCTLR_EL3.TMT;
    otherwise
        Unreachable();

boolean enable = tme == '1';
boolean trivial = tmt == '1';

if !enable then
    TransactionStartTrap(t);
elsif trivial then
    TSTATE.nPC = NextInstrAddr(64);
```

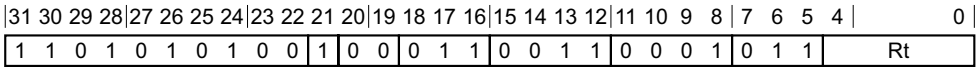
```
    TSTATE.Rt = t;
    FailTransaction(TMFailure_TRIVIAL, FALSE);
elsif IsFeatureImplemented(FEAT_SME) && PSTATE.SM == '1' then
    FailTransaction(TMFailure_ERR, FALSE);
elsif TSTATE.depth == 255 then
    FailTransaction(TMFailure_NEST, FALSE);
elsif TSTATE.depth == 0 then
    TSTATE.nPC = NextInstrAddr(64);
    TSTATE.Rt = t;
    ClearExclusiveLocal(ProcessorID());
    TakeTransactionCheckpoint();
    StartTrackingTransactionalReadsWrites();

TSTATE.depth = TSTATE.depth + 1;
X[t, 64] = Zeros(64);
```

C6.2.430 TTEST

This instruction writes the depth of the transaction to the destination register, or the value 0 otherwise.

System

(FEAT_TME)



Encoding

TTEST <Xt>

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_TME) then UNDEFINED;
integer t = UInt(Rt);
```

Assembler symbols

<Xt> Is the 64-bit name of the general-purpose destination register, encoded in the "Rt" field.

Operation

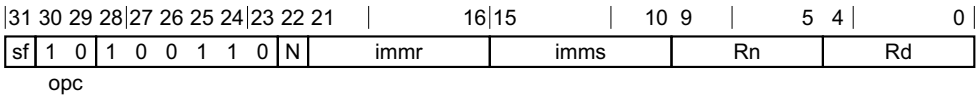
```
if !IsTMEEnabled() then UNDEFINED;

X[t, 64] = (TSTATE.depth)<63:0>;
```

C6.2.431 UBFIZ

Unsigned Bitfield Insert in Zeros copies a bitfield of <width> bits from the least significant bits of the source register to bit position <lsb> of the destination register, setting the destination bits above and below the bitfield to zero.

This instruction is an alias of the UBFM instruction. This means that:

- The encodings in this description are named to match the encodings of UBFM.
- The description of UBFM gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.



32-bit variant

Applies when `sf == 0 && N == 0`.

```
UBFIZ <Wd>, <Wn>, #<lsb>, #<width>
```

is equivalent to

```
UBFM <Wd>, <Wn>, #(-<lsb> MOD 32), #(<width>-1)
```

and is the preferred disassembly when `UInt(imms) < UInt(immr)`.

64-bit variant

Applies when `sf == 1 && N == 1`.

```
UBFIZ <Xd>, <Xn>, #<lsb>, #<width>
```

is equivalent to

```
UBFM <Xd>, <Xn>, #(-<lsb> MOD 64), #(<width>-1)
```

and is the preferred disassembly when `UInt(imms) < UInt(immr)`.

Assembler symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
<lsb>	For the 32-bit variant: is the bit number of the lsb of the destination bitfield, in the range 0 to 31. For the 64-bit variant: is the bit number of the lsb of the destination bitfield, in the range 0 to 63.
<width>	For the 32-bit variant: is the width of the bitfield, in the range 1 to 32-<lsb>. For the 64-bit variant: is the width of the bitfield, in the range 1 to 64-<lsb>.

Operation

The description of UBFM gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

C6.2.432 UBFM

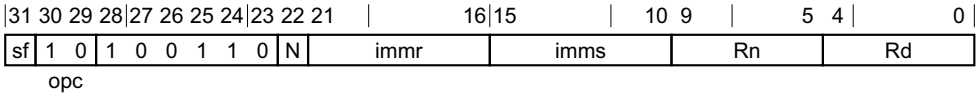
Unsigned Bitfield Move is usually accessed via one of its aliases, which are always preferred for disassembly.

If <imms> is greater than or equal to <immr>, this copies a bitfield of (<imms>-<immr>+1) bits starting from bit position <immr> in the source register to the least significant bits of the destination register.

If <imms> is less than <immr>, this copies a bitfield of (<imms>+1) bits from the least significant bits of the source register to bit position (regsize-<immr>) of the destination register, where regsize is the destination register size of 32 or 64 bits.

In both cases the destination bits below and above the bitfield are set to zero.

This instruction is used by the aliases LSL (immediate), LSR (immediate), UBFIZ, UBFX, UXTB, and UXTH. See *Alias conditions* for details of when each alias is preferred.



32-bit variant

Applies when sf == 0 && N == 0.

```
UBFM <Wd>, <Wn>, #<immr>, #<imms>
```

64-bit variant

Applies when sf == 1 && N == 1.

```
UBFM <Xd>, <Xn>, #<immr>, #<imms>
```

Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
constant integer datasize = 32 << UInt(sf);

integer r;
bits(datasize) wmask;
bits(datasize) tmask;

if sf == '1' && N != '1' then UNDEFINED;
if sf == '0' && (N != '0' || immr<5> != '0' || imms<5> != '0') then UNDEFINED;

r = UInt(immr);
(wmask, tmask) = DecodeBitMasks(N, imms, immr, FALSE, datasize);
```

Alias conditions

Alias	of variant	is preferred when
LSL (immediate)	32-bit	`imms != '011111' && imms + 1 == immr`
LSL (immediate)	64-bit	`imms != '111111' && imms + 1 == immr`
LSR (immediate)	32-bit	`imms == '011111'`
LSR (immediate)	64-bit	`imms == '111111'`
UBFIZ	-	`UInt(imms) < UInt(immr)`
UBFX	-	`BFXPreferred(sf, opc<1>, imms, immr)`
UXTB	-	`immr == '000000' && imms == '000111'`
UXTH	-	`immr == '000000' && imms == '001111'`

Assembler symbols

`<Wd>` Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

`<Wn>` Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.

`<Xd>` Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

`<Xn>` Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.

`<immr>` For the 32-bit variant: is the right rotate amount, in the range 0 to 31, encoded in the "immr" field.

For the 64-bit variant: is the right rotate amount, in the range 0 to 63, encoded in the "immr" field.

`<imms>` For the 32-bit variant: is the leftmost bit number to be moved from the source, in the range 0 to 31, encoded in the "imms" field.

For the 64-bit variant: is the leftmost bit number to be moved from the source, in the range 0 to 63, encoded in the "imms" field.

Operation

```
bits(datasize) src = X[n, datasize];

// perform bitfield move on low bits
bits(datasize) bot = ROR(src, r) AND wmask;

// combine extension bits and result bits
X[d, datasize] = bot AND tmask;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.

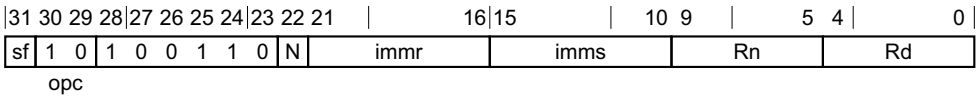
— The values of the NZCV flags.

C6.2.433 UBFX

Unsigned Bitfield Extract copies a bitfield of <width> bits starting from bit position <lsb> in the source register to the least significant bits of the destination register, and sets destination bits above the bitfield to zero.

This instruction is an alias of the UBFM instruction. This means that:

- The encodings in this description are named to match the encodings of UBFM.
- The description of UBFM gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.



32-bit variant

Applies when `sf == 0 && N == 0`.

`UBFX <Wd>, <Wn>, #<lsb>, #<width>`

is equivalent to

`UBFM <Wd>, <Wn>, #<lsb>, #(<lsb>+<width>-1)`

and is the preferred disassembly when `BFXPreferred(sf, opc<1>, imms, immr)`.

64-bit variant

Applies when `sf == 1 && N == 1`.

`UBFX <Xd>, <Xn>, #<lsb>, #<width>`

is equivalent to

`UBFM <Xd>, <Xn>, #<lsb>, #(<lsb>+<width>-1)`

and is the preferred disassembly when `BFXPreferred(sf, opc<1>, imms, immr)`.

Assembler symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
<lsb>	For the 32-bit variant: is the bit number of the lsb of the source bitfield, in the range 0 to 31. For the 64-bit variant: is the bit number of the lsb of the source bitfield, in the range 0 to 63.
<width>	For the 32-bit variant: is the width of the bitfield, in the range 1 to 32-<lsb>. For the 64-bit variant: is the width of the bitfield, in the range 1 to 64-<lsb>.

Operation

The description of UBFM gives the operational pseudocode for this instruction.

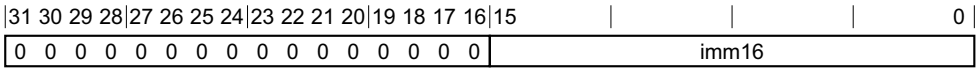
Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

C6.2.434 UDF

Permanently Undefined generates an Undefined Instruction exception (ESR_ELx.EC = 0b000000). The encodings for UDF used in this section are defined as permanently UNDEFINED.



Encoding

```
UDF #<imm>
```

Decode for this encoding

```
// The imm16 field is ignored by hardware.
UNDEFINED;
```

Assembler symbols

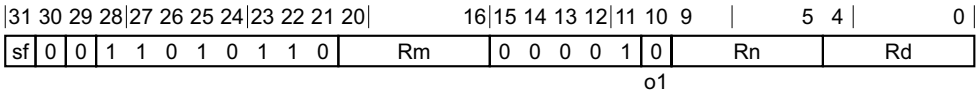
<imm> is a 16-bit unsigned immediate, in the range 0 to 65535, encoded in the "imm16" field. The PE ignores the value of this constant.

Operation

```
// No operation.
```

C6.2.435 UDIV

Unsigned Divide divides an unsigned integer register value by another unsigned integer register value, and writes the result to the destination register. The condition flags are not affected.



32-bit variant

Applies when `sf == 0`.

```
UDIV <Wd>, <Wn>, <Wm>
```

64-bit variant

Applies when `sf == 1`.

```
UDIV <Xd>, <Xn>, <Xm>
```

Decode for all variants of this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
constant integer datasize = 32 << UInt(sf);
```

Assembler symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Wm>	Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.

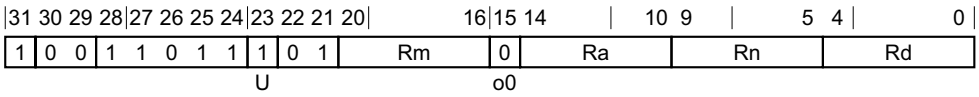
Operation

```
bits(datasize) operand1 = X[n, datasize];
bits(datasize) operand2 = X[m, datasize];
integer result;

if IsZero(operand2) then
    result = 0;
else
    result = RoundTowardsZero(Real(Int(operand1, TRUE)) / Real(Int(operand2, TRUE)));

X[d, datasize] = result<datasize-1:0>;
```

C6.2.436 UMADDL

Unsigned Multiply-Add Long multiplies two 32-bit register values, adds a 64-bit register value, and writes the result to the 64-bit destination register.

This instruction is used by the alias UMULL. See *Alias conditions* for details of when each alias is preferred.



Encoding

```
UMADDL <Xd>, <Wn>, <Wm>, <Xa>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer a = UInt(Ra);
```

Alias conditions

Alias	is preferred when
UMULL	`Ra == '11111'`

Assembler symbols

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Wn> Is the 32-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.

<Wm> Is the 32-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.

<Xa> Is the 64-bit name of the third general-purpose source register holding the addend, encoded in the "Ra" field.

Operation

```
bits(32) operand1 = X[n, 32];
bits(32) operand2 = X[m, 32];
bits(64) operand3 = X[a, 64];

integer result;

result = Int(operand3, TRUE) + (Int(operand1, TRUE) * Int(operand2, TRUE));

X[d, 64] = result<63:0>;
```

Operational information

If PSTATE.DIT is 1:

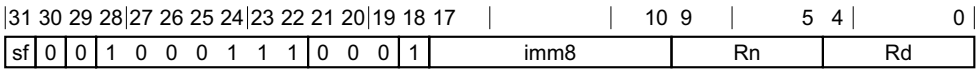
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

C6.2.437 UMAX (immediate)

Unsigned Maximum (immediate) determines the unsigned maximum of the source register value and immediate, and writes the result to the destination register.

Integer

(FEAT_CSSC)



32-bit variant

Applies when `sf == 0`.

```
UMAX <Wd>, <Wn>, #<uimm>
```

64-bit variant

Applies when `sf == 1`.

```
UMAX <Xd>, <Xn>, #<uimm>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_CSSC) then UNDEFINED;
constant integer datasize = 32 << UInt(sf);
integer n = UInt(Rn);
integer d = UInt(Rd);
integer imm = UInt(imm8);
```

Assembler symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
<uimm>	Is an unsigned immediate, in the range 0 to 255, encoded in the "imm8" field.

Operation

```
bits(datasize) operand1 = X[n, datasize];
integer result = Max(UInt(operand1), imm);
X[d, datasize] = result<datasize-1:0>;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

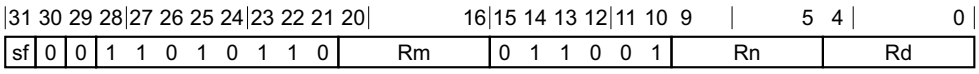
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

C6.2.438 UMAX (register)

Unsigned Maximum (register) determines the unsigned maximum of the two source register values and writes the result to the destination register.

Integer

(FEAT_CSSC)



32-bit variant

Applies when `sf == 0`.

```
UMAX <Wd>, <Wn>, <Wm>
```

64-bit variant

Applies when `sf == 1`.

```
UMAX <Xd>, <Xn>, <Xm>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_CSSC) then UNDEFINED;
constant integer datasize = 32 << UInt(sf);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer d = UInt(Rd);
```

Assembler symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Wm>	Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.

Operation

```
bits(datasize) operand1 = X[n, datasize];
bits(datasize) operand2 = X[m, datasize];
integer result = Max(UInt(operand1), UInt(operand2));
X[d, datasize] = result<datasize-1:0>;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.

  - The values of the NZCV flags.

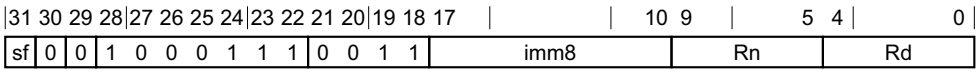
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

C6.2.439 UMIN (immediate)

Unsigned Minimum (immediate) determines the unsigned minimum of the source register value and immediate, and writes the result to the destination register.

Integer

(FEAT_CSSC)



32-bit variant

Applies when `sf == 0`.

```
UMIN <Wd>, <Wn>, #<uimm>
```

64-bit variant

Applies when `sf == 1`.

```
UMIN <Xd>, <Xn>, #<uimm>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_CSSC) then UNDEFINED;
constant integer datasize = 32 << UInt(sf);
integer n = UInt(Rn);
integer d = UInt(Rd);
integer imm = UInt(imm8);
```

Assembler symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
<uimm>	Is an unsigned immediate, in the range 0 to 255, encoded in the "imm8" field.

Operation

```
bits(datasize) operand1 = X[n, datasize];
integer result = Min(UInt(operand1), imm);
X[d, datasize] = result<datasize-1:0>;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

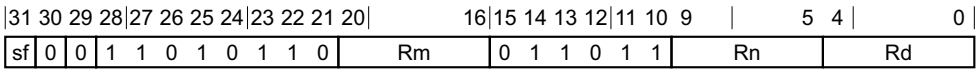
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

C6.2.440 UMIN (register)

Unsigned Minimum (register) determines the unsigned minimum of the two source register values and writes the result to the destination register.

Integer

(FEAT_CSSC)



32-bit variant

Applies when `sf == 0`.

```
UMIN <Wd>, <Wn>, <Wm>
```

64-bit variant

Applies when `sf == 1`.

```
UMIN <Xd>, <Xn>, <Xm>
```

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_CSSC) then UNDEFINED;
constant integer datasize = 32 << UInt(sf);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer d = UInt(Rd);
```

Assembler symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Wm>	Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.

Operation

```
bits(datasize) operand1 = X[n, datasize];
bits(datasize) operand2 = X[m, datasize];
integer result = Min(UInt(operand1), UInt(operand2));
X[d, datasize] = result<datasize-1:0>;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.

  - The values of the NZCV flags.

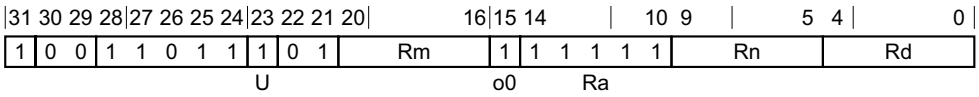
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

C6.2.441 UMNEGL

Unsigned Multiply-Negate Long multiplies two 32-bit register values, negates the product, and writes the result to the 64-bit destination register.

This instruction is an alias of the UMSUBL instruction. This means that:

- The encodings in this description are named to match the encodings of UMSUBL.
- The description of UMSUBL gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.



Encoding

```
UMNEGL <Xd>, <Wn>, <Wm>
```

is equivalent to

```
UMSUBL <Xd>, <Wn>, <Wm>, XZR
```

and is always the preferred disassembly.

Assembler symbols

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Wn> Is the 32-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.

<Wm> Is the 32-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.

Operation

The description of UMSUBL gives the operational pseudocode for this instruction.

Operational information

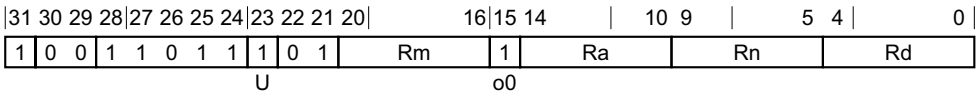
If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

C6.2.442 UMSUBL

Unsigned Multiply-Subtract Long multiplies two 32-bit register values, subtracts the product from a 64-bit register value, and writes the result to the 64-bit destination register.

This instruction is used by the alias UMNEGL. See *Alias conditions* for details of when each alias is preferred.



Encoding

```
UMSUBL <Xd>, <Wn>, <Wm>, <Xa>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer a = UInt(Ra);
```

Alias conditions

Alias	is preferred when
UMNEGL	`Ra == '11111'`

Assembler symbols

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Wn> Is the 32-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.

<Wm> Is the 32-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.

<Xa> Is the 64-bit name of the third general-purpose source register holding the minuend, encoded in the "Ra" field.

Operation

```
bits(32) operand1 = X[n, 32];
bits(32) operand2 = X[m, 32];
bits(64) operand3 = X[a, 64];

integer result;

result = Int(operand3, TRUE) - (Int(operand1, TRUE) * Int(operand2, TRUE));
X[d, 64] = result<63:0>;
```

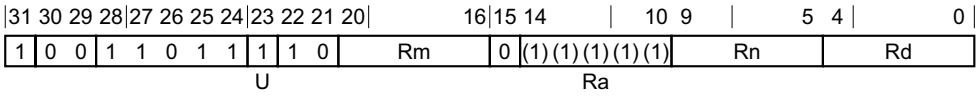
Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

C6.2.443 UMULH

Unsigned Multiply High multiplies two 64-bit register values, and writes bits[127:64] of the 128-bit result to the 64-bit destination register.



Encoding

```
UMULH <Xd>, <Xn>, <Xm>
```

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
```

Assembler symbols

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Xn> Is the 64-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.

<Xm> Is the 64-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.

Operation

```
bits(64) operand1 = X[n, 64];
bits(64) operand2 = X[m, 64];

integer result;

result = Int(operand1, TRUE) * Int(operand2, TRUE);

X[d, 64] = result<127:64>;
```

Operational information

If PSTATE.DIT is 1:

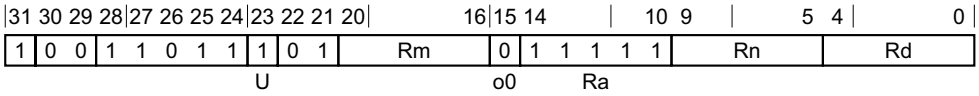
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

C6.2.444 UMULL

Unsigned Multiply Long multiplies two 32-bit register values, and writes the result to the 64-bit destination register.

This instruction is an alias of the UMADDL instruction. This means that:

- The encodings in this description are named to match the encodings of UMADDL.
- The description of UMADDL gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.



Encoding

```
UMULL <Xd>, <Wn>, <Wm>
```

is equivalent to

```
UMADDL <Xd>, <Wn>, <Wm>, XZR
```

and is always the preferred disassembly.

Assembler symbols

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Wn> Is the 32-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.

<Wm> Is the 32-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.

Operation

The description of UMADDL gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

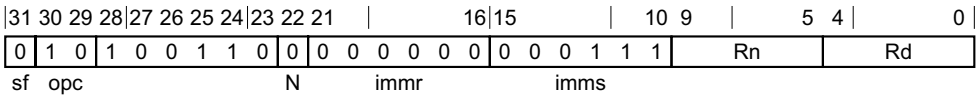
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

C6.2.445 UXTB

Unsigned Extend Byte extracts an 8-bit value from a register, zero-extends it to the size of the register, and writes the result to the destination register.

This instruction is an alias of the UBFM instruction. This means that:

- The encodings in this description are named to match the encodings of UBFM.
- The description of UBFM gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.



32-bit variant

```
UXTB <Wd>, <Wn>
```

is equivalent to

```
UBFM <Wd>, <Wn>, #0, #7
```

and is always the preferred disassembly.

Assembler symbols

`<Wd>` Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

`<Wn>` Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.

Operation

The description of UBFM gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

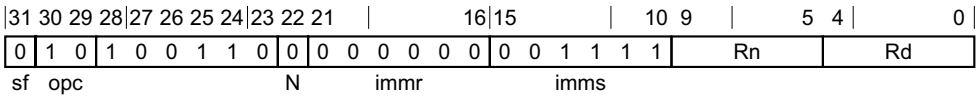
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

C6.2.446 UXTH

Unsigned Extend Halfword extracts a 16-bit value from a register, zero-extends it to the size of the register, and writes the result to the destination register.

This instruction is an alias of the UBFM instruction. This means that:

- The encodings in this description are named to match the encodings of UBFM.
- The description of UBFM gives the operational pseudocode, any CONSTRAINED UNPREDICTABLE behavior, and any operational information for this instruction.



32-bit variant

```
UXTH <Wd>, <Wn>
```

is equivalent to

```
UBFM <Wd>, <Wn>, #0, #15
```

and is always the preferred disassembly.

Assembler symbols

`<Wd>` Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

`<Wn>` Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.

Operation

The description of UBFM gives the operational pseudocode for this instruction.

Operational information

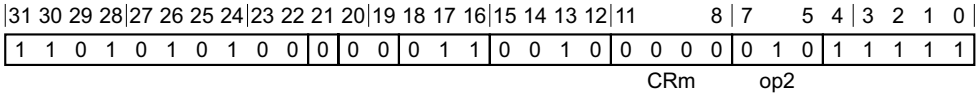
If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

C6.2.447 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*.

As described in *Wait for 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.



Encoding

WFE

Decode for this encoding

```
// Empty.
```

Operation

```
integer localtimeout = 1 << 64;    // No local timeout event is generated
Hint_WFE(localtimeout, WFxType_WFE);
```

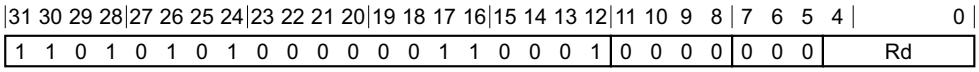
C6.2.448 WFET

Wait For Event with Timeout is a hint instruction that indicates that the PE can enter a low-power state and remain there until either a local timeout event or 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*.

As described in *Wait for Event*, the execution of a WFET instruction that would otherwise cause entry to a low-power state can be trapped to a higher Exception level.

System

(FEAT_WFxT)



Encoding

WFET <Xt>

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_WFxT) then UNDEFINED;

integer d = UInt(Rd);
```

Assembler symbols

<Xt> Is the 64-bit name of the general-purpose source register, encoded in the "Rd" field.

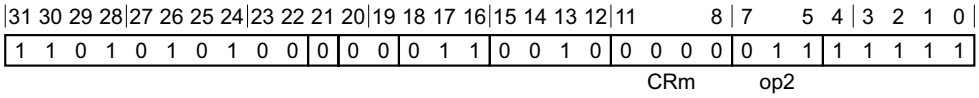
Operation

```
integer localtimeout = UInt(X[d, 64]);

if Halted() && ConstrainUnpredictableBool(Unpredictable_WFxTDEBUG) then
    EndOfInstruction();

Hint_WFE(localtimeout, WFxType_WFET);
```

C6.2.449 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 mechanism*.

As described in *Wait for Interrupt mechanism*, the execution of a `WFI` instruction that would otherwise cause entry to a low-power state can be trapped to a higher Exception level.



Encoding

`WFI`

Decode for this encoding

```
// Empty.
```

Operation

```
integer localtimeout = 1 << 64;    // No local timeout event is generated
Hint_WFI(localtimeout, WFxType_WFI);
```

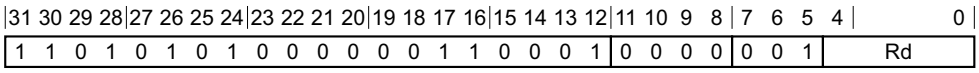
C6.2.450 WFIT

Wait For Interrupt with Timeout is a hint instruction that indicates that the PE can enter a low-power state and remain there until either a local timeout event or a wakeup event occurs. For more information, see *Wait for Interrupt mechanism*.

As described in *Wait for Interrupt mechanism*, the execution of a `WFIT` instruction that would otherwise cause entry to a low-power state can be trapped to a higher Exception level.

System

(FEAT_WFxT)



Encoding

`WFIT <Xt>`

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_WFxT) then UNDEFINED;

integer d = UInt(Rd);
```

Assembler symbols

`<Xt>` Is the 64-bit name of the general-purpose source register, encoded in the "Rd" field.

Operation

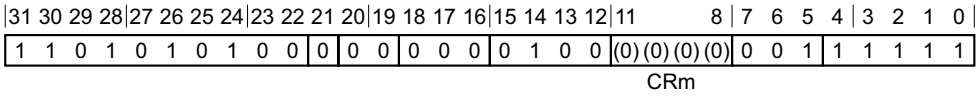
```
integer localtimeout = UInt(X[d, 64]);

if Halted() && ConstrainUnpredictableBool(Unpredictable_WFxTDEBUG) then
    EndOfInstruction();

Hint_WFI(localtimeout, WFxType_WFIT);
```

C6.2.451 XAFLAG

Convert floating-point condition flags from external format to Arm format. This instruction converts the state of the PSTATE.{N,Z,C,V} flags from an alternative representation required by some software to a form representing the result of an Arm floating-point scalar compare instruction.

System

(FEAT_FlagM2)



Encoding

XAFLAG

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_FlagM2) then UNDEFINED;
```

Operation

```
bit n = NOT(PSTATE.C) AND NOT(PSTATE.Z);
bit z = PSTATE.Z AND PSTATE.C;
bit c = PSTATE.C OR PSTATE.Z;
bit v = NOT(PSTATE.C) AND PSTATE.Z;

PSTATE.N = n;
PSTATE.Z = z;
PSTATE.C = c;
PSTATE.V = v;
```

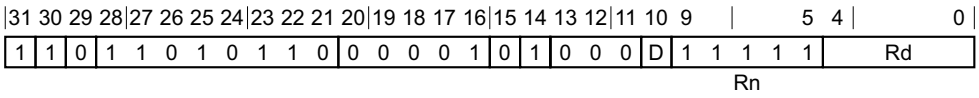
C6.2.452 XPACD, XPACI, XPACLRI

Strip Pointer Authentication Code. This instruction removes the pointer authentication code from an address. The address is in the specified general-purpose register for XPACI and XPACD, and is in LR for XPACLRI.

The XPACD instruction is used for data addresses, and XPACI and XPACLRI are used for instruction addresses.

Integer

(FEAT_PAuth)



XPACD variant

Applies when D == 1.

```
XPACD <Xd>
```

XPACI variant

Applies when D == 0.

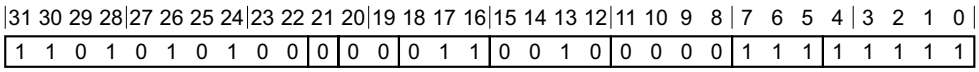
```
XPACI <Xd>
```

Decode for all variants of this encoding

```
boolean data = (D == '1');
integer d = UInt(Rd);

if !IsFeatureImplemented(FEAT_PAuth) then
    UNDEFINED;
```

System

(FEAT_PAuth)



Encoding

```
XPACLRI
```

Decode for this encoding

```
integer d = 30;
boolean data = FALSE;
```

Assembler symbols

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

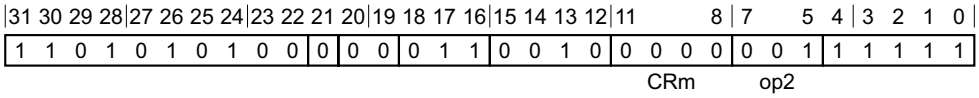
Operation for all encodings

```
if IsFeatureImplemented(FEAT_PAuth) then
    X[d, 64] = Strip(X[d, 64], data);
```

C6.2.453 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*.



Encoding

`YIELD`

Decode for this encoding

```
// Empty.
```

Operation

```
Hint_Yield();
```