C6.2.1 ADC

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

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

32-bit 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);
integer datasize = if sf == '1' then 64 else 32;
```

Assembler symbols

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

Operation

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

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

X[d] = result;
```

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.2 ADCS

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

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

32-bit 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);
integer datasize = if sf == '1' then 64 else 32;
```

Assembler symbols

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

Operation

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

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

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

X[d] = result;
```

C6.2.3 ADD (extended register)

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

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

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);
integer datasize = if sf == '1' then 64 else 32;
ExtendType extend_type = DecodeRegExtend(option);
integer shift = UInt(imm3);
if shift > 4 then ReservedValue();
```

Assembler symbols

`<Wd|WSP>`	Is the 32-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.
`<Wn|WSP>`	Is the 32-bit name of the first source general-purpose register or stack pointer, encoded in the "Rn" field.
`<Wm>`	Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
`<Xd|SP>`	Is the 64-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.
`<Xn|SP>`	Is the 64-bit name of the first source general-purpose register or stack pointer, encoded in the "Rn" field.

`<R>` Is a width specifier, encoded in 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[] else X[n];
bits(datasize) operand2 = ExtendReg(m, extend_type, shift);

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

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

C6.2.4 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.

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

32-bit 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);
integer datasize = if sf == '1' then 64 else 32;
bits(datasize) imm;

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

Alias conditions

Alias	is preferred when
MOV (to/from SP)	`shift == '00' && 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.

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

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

`LSL #0`	when `shift = 00`
`LSL #12`	when `shift = 01`

The encoding `shift = 1x` is reserved.

Operation

```
bits(datasize) result;
bits(datasize) operand1 = if n == 31 then SP[] else X[n];

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

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

C6.2.5 ADD (shifted register)

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

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

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);
integer datasize = if sf == '1' then 64 else 32;

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

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

Assembler symbols

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

`<shift>` Is the optional shift type to be applied to the second source operand, defaulting to LSL and encoded in 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.

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

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

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

X[d] = result;
```

C6.2.6 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.

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

32-bit 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);
integer datasize = if sf == '1' then 64 else 32;
ExtendType extend_type = DecodeRegExtend(option);
integer shift = UInt(imm3);
if shift > 4 then ReservedValue();
```

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`

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

`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[] else X[n];
bits(datasize) operand2 = ExtendReg(m, extend_type, shift);
bits(4) nzcv;

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

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

X[d] = result;
```

C6.2.7 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.

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

32-bit 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);
integer datasize = if sf == '1' then 64 else 32;
bits(datasize) imm;

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

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 "shift" field. It can have the following values:

`LSL #0`	when `shift = 00`
`LSL #12`	when `shift = 01`

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

The encoding `shift = 1x` is reserved.

Operation

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

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

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

X[d] = result;
```

C6.2.8 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.

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

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);
integer datasize = if sf == '1' then 64 else 32;

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

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.

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

`<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];
bits(datasize) operand2 = ShiftReg(m, shift_type, shift_amount);
bits(4) nzcv;

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

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

X[d] = result;
```

C6.2.9 ADR

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

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

Literal variant

```
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".

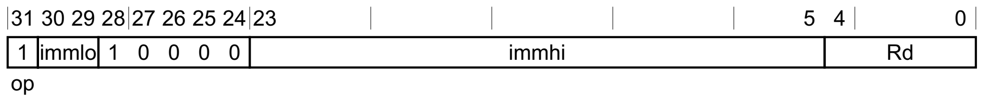
Operation

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

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

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.10 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.



Literal variant

```
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 = PC[];

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

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

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.11 AND (immediate)

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

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

32-bit 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);
integer datasize = if sf == '1' then 64 else 32;
bits(datasize) imm;
if sf == '0' && N != '0' then ReservedValue();
(imm, -) = DecodeBitMasks(N, imms, immr, TRUE);
```

Assembler symbols

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

Operation

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

result = operand1 AND imm;
if d == 31 then
    SP[] = result;
else
    X[d] = result;
```

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.12 AND (shifted register)

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

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

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);
integer datasize = if sf == '1' then 64 else 32;
if sf == '0' && imm6<5> == '1' then ReservedValue();

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

Assembler symbols

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

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

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

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

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.13 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.

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

32-bit 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);
integer datasize = if sf == '1' then 64 else 32;

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

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];
```

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

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

X[d] = result;
```

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.14 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.

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

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);
integer datasize = if sf == '1' then 64 else 32;

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

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`

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

`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];
bits(datasize) operand2 = ShiftReg(m, shift_type, shift_amount);

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

X[d] = result;
```

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.15 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 for this instruction.

31	30	29	28 27 26 25 24 23 22 21	20 16	15 14 13 12	11 10	9 5	4 0
sf	0	0	1 1 0 1 0 1 1 0	Rm	0 0 1 0	1 0	Rn	Rd
						op2		

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.

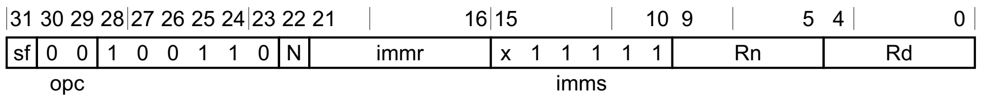
Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.16 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 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.

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.17 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.

31	30	29	28	27	26	25	24	23	22	21	20–16	15	14	13	12	11	10	9–5	4–0
sf	0	0	1	1	0	1	0	1	1	0	Rm	0	0	1	0	1	0	Rn	Rd
																op2			

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);
integer datasize = if sf == '1' then 64 else 32;
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

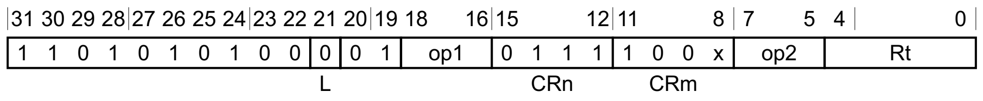
```
bits(datasize) result;
bits(datasize) operand2 = X[m];

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

C6.2.18 AT

Address Translate. For more information, see A64 system instructions for address translation.

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 for this instruction.



System variant

```
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 ARMv8.2-ATS1E1 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

`<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.

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

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

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.19 B.cond

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

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

19-bit signed PC-relative branch offset variant

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

Decode for this encoding

```
bits(64) offset = SignExtend(imm19:'00', 64);
bits(4) condition = cond;
```

Assembler symbols

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

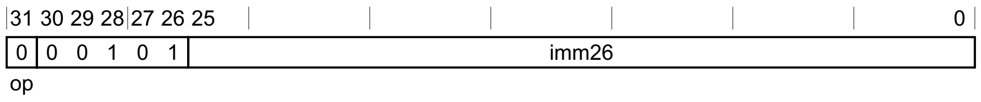
Operation

```
if ConditionHolds(condition) then
    BranchTo(PC[] + offset, BranchType_JMP);
```

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.20 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.



26-bit signed PC-relative branch offset variant

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

C6.2.21 BFC

Bitfield Clear, leaving other 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 for this instruction.

ARMv8.2

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

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.

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.22 BFI

Bitfield Insert copies any number of low-order bits from a source register into the same number of adjacent bits at any position in the destination register, leaving other 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 for this instruction.

31	30	29	28	27	26	25	24	23	22	21 16	15 10	9 5	4 0
sf	0	1	1	0	0	1	1	0	N	immr	imms	!=11111	Rd
	opc											Rn	

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>.

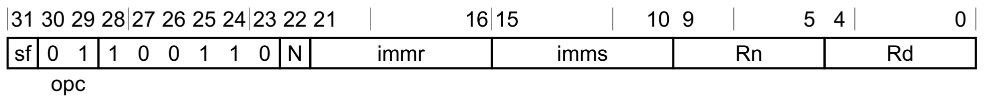
Operation

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

C6.2.23 BFM

Bitfield Move copies any number of low-order bits from a source register into the same number of adjacent bits at any position in the destination register, leaving other bits 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);
integer datasize = if sf == '1' then 64 else 32;

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

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

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

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

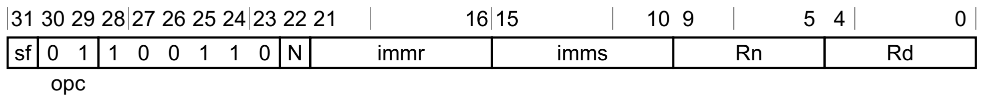
```
bits(datasize) dst = X[d];
bits(datasize) src = X[n];

// 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] = (dst AND NOT(tmask)) OR (bot AND tmask);
```

C6.2.24 BFXIL

Bitfield extract and insert at low end copies any number of low-order bits from a source register into the same number of adjacent bits at the low end in the destination register, leaving other 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 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.

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.25 BIC (shifted register)

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

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

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);
integer datasize = if sf == '1' then 64 else 32;
if sf == '0' && imm6<5> == '1' then ReservedValue();

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

Assembler symbols

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

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

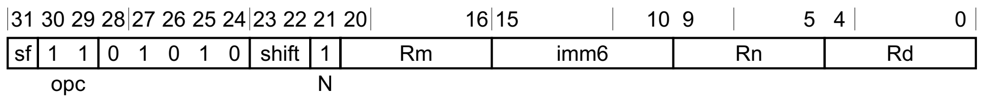
Operation

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

operand2 = NOT(operand2);

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

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.26 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);
integer datasize = if sf == '1' then 64 else 32;

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

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

Assembler symbols

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

`<shift>` Is the optional shift 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,

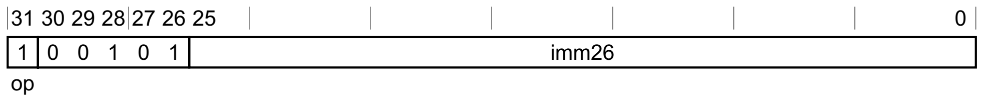
Operation

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

operand2 = NOT(operand2);

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

X[d] = result;
```

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.27 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.



26-bit signed PC-relative branch offset variant

```
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

```
X[30] = PC[] + 4;

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

C6.2.28 BLR

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

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

Integer variant

```
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];

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

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.29 BR

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

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

Integer variant

```
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];

BranchTo(target, BranchType_JMP);
```

C6.2.30 BRK

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

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

System variant

```
BRK #<imm>
```

Decode for this encoding

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

Assembler symbols

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

Operation

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

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.31 CASB, CASAB, CASALB, CASLB

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

- `CASAB` and `CASALB` load from memory with acquire semantics.
- `CASLB` and `CASALB` store to memory with release semantics.
- `CASB` has no memory ordering requirements.

For more information about memory ordering semantics see *Load-Acquire, Store-Release* on page B2-94.

For information about memory accesses see *Load/Store addressing modes* on page C1-143.

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.

ARMv8.1

31 30	29 28 27 26 25 24	23	22	21	20 ... 16	15	14 13 12 11 10	9 ... 5	4 ... 0
0 0	0 0 1 0 0 0	1	L	1	Rs	o0	1 1 1 1 1	Rn	Rt
size									

Acquire variant

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

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

Acquire and release variant

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

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

No memory ordering variant

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

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

Release variant

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

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

Decode for all variants of this encoding

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

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

AccType ldacctype = if L == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
AccType stacctype = if o0 == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
```

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

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;

comparevalue = X[s];
newvalue = X[t];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, 1, ldacctype];
if data == comparevalue then
    // All observers in the shareability domain observe the
    // following load and store atomically.
    Mem[address, 1, stacctype] = newvalue;

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

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.32 CASH, CASAH, CASALH, CASLH

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

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

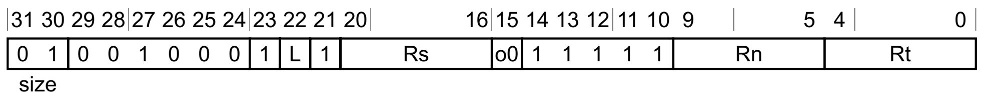
For more information about memory ordering semantics see *Load-Acquire, Store-Release* on page B2-94.

For information about memory accesses see *Load/Store addressing modes* on page C1-143.

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.

ARMv8.1



Acquire variant

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

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

Acquire and release variant

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

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

No memory ordering variant

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

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

Release variant

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

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

Decode for all variants of this encoding

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

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

AccType ldacctype = if L == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
AccType stacctype = if o0 == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
```

Assembler symbols

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

Operation

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

comparevalue = X[s];
newvalue = X[t];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, 2, ldacctype];
if data == comparevalue then
    // All observers in the shareability domain observe the
    // following load and store atomically.
    Mem[address, 2, stacctype] = newvalue;

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

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.33 CASP, CASPA, CASPAL, CASPL

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

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

For more information about memory ordering semantics see *Load-Acquire, Store-Release* on page B2-94.

For information about memory accesses see *Load/Store addressing modes* on page C1-143.

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.

ARMv8.1

31	30	29	28	27	26	25	24	23	22	21	20 16	15	14				10	9 5	4 0
0	sz	0	0	1	0	0	0	0	L	1	Rs	o0	1	1	1	1	1	Rn	Rt
															Rt2				

32-bit, acquire variant

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

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

32-bit, acquire and release variant

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

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

32-bit, no memory ordering variant

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

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

32-bit, release variant

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

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

64-bit, acquire variant

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

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

64-bit, acquire and release variant

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

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

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

64-bit, no memory ordering variant

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

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

64-bit, release 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 !HaveAtomicExt() then UnallocatedEncoding();
if Rs<0> == '1' then UnallocatedEncoding();
if Rt<0> == '1' then UnallocatedEncoding();

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

integer datasize = 32 << UInt(sz);
AccType ldacctype = if L == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
AccType stacctype = if o0 == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
```

Assembler symbols

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

Operation

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

bits(datasize) s1 = X[s];
bits(datasize) s2 = X[s+1];
bits(datasize) t1 = X[t];
bits(datasize) t2 = X[t+1];
comparevalue = if BigEndian() then s1:s2 else s2:s1;
newvalue = if BigEndian() then t1:t2 else t2:t1;
```

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

```
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, (2*datasize) DIV 8, ldacctype];
if data == comparevalue then
    // All observers in the shareability domain observe the
    // following load and store atomically.
    Mem[address, (2*datasize) DIV 8, stacctype] = newvalue;

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

C6.2.34 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 no memory ordering requirements.

For more information about memory ordering semantics see *Load-Acquire, Store-Release* on page B2-94.

For information about memory accesses see *Load/Store addressing modes* on page C1-143.

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.

ARMv8.1

31	30	29	28	27	26	25	24	23	22	21	20 16	15	14	13	12	11	10	9 5	4 0
1	x	0	0	1	0	0	0	1	L	1	Rs	o0	1	1	1	1	1	Rn	Rt
size																			

32-bit, acquire variant

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

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

32-bit, acquire and release variant

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

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

32-bit, no memory ordering variant

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

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

32-bit, release variant

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

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

64-bit, acquire variant

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

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

64-bit, acquire and release variant

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

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

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

64-bit, no memory ordering variant

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

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

64-bit, release variant

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

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

Decode for all variants of this encoding

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

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

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if L == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
AccType stacctype = if o0 == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
```

Assembler symbols

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

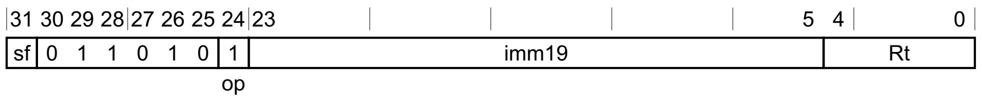
Operation

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

comparevalue = X[s];
newvalue = X[t];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, datasize DIV 8, ldacctype];
if data == comparevalue then
    // All observers in the shareability domain observe the
    // following load and store atomically.
    Mem[address, datasize DIV 8, stacctype] = newvalue;

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

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.35 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);
integer datasize = if sf == '1' then 64 else 32;
bits(64) offset = SignExtend(imm19:'00', 64);
```

Assembler symbols

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

Operation

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

if IsZero(operand1) == FALSE then
    BranchTo(PC[] + offset, BranchType_JMP);
```

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.36 CBZ

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

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

32-bit 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);
integer datasize = if sf == '1' then 64 else 32;
bits(64) offset = SignExtend(imm19:'00', 64);
```

Assembler symbols

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

Operation

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

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

C6.2.37 CCMN (immediate)

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

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

32-bit 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);
integer datasize = if sf == '1' then 64 else 32;
bits(4) flags = nzcv;
bits(datasize) imm = ZeroExtend(imm5, datasize);
```

Assembler symbols

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

Operation

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

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

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.38 CCMN (register)

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

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

32-bit 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);
integer datasize = if sf == '1' then 64 else 32;
bits(4) flags = nzcv;
```

Assembler symbols

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

Operation

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

if ConditionHolds(cond) then
    (-, flags) = AddWithCarry(operand1, operand2, '0');
PSTATE.<N,Z,C,V> = flags;
```

C6.2.39 CCMP (immediate)

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

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

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);
integer datasize = if sf == '1' then 64 else 32;
bits(4) flags = nzcv;
bits(datasize) imm = ZeroExtend(imm5, datasize);
```

Assembler symbols

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

Operation

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

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

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.40 CCMP (register)

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

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

32-bit 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);
integer datasize = if sf == '1' then 64 else 32;
bits(4) flags = nzcv;
```

Assembler symbols

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

Operation

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

if ConditionHolds(cond) then
    operand2 = NOT(operand2);
    (-, flags) = AddWithCarry(operand1, operand2, '1');
PSTATE.<N,Z,C,V> = flags;
```

C6.2.41 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 for this instruction.

31	30	29	28	27	26	25	24	23	22	21	20 16	15 12	11	10	9 5	4 0
sf	0	0	1	1	0	1	0	1	0	0	!=11111	!=111x	0	1	!=11111	Rd
	op										Rm	cond		o2	Rn	

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.

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.42 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 for this instruction.

31	30	29	28	27	26	25	24	23	22	21	20 16	15 12	11	10	9 5	4 0
sf	1	0	1	1	0	1	0	1	0	0	!=11111	!=111x	0	0	!=11111	Rd
	op										Rm	cond		o2	Rn	

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.

C6.2.43 CLREX

Clear Exclusive clears the local monitor of the executing PE.

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

System variant

```
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());
```

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.44 CLS

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

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

32-bit 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);
integer datasize = if sf == '1' then 64 else 32;
```

Assembler symbols

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

Operation

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

result = CountLeadingSignBits(operand1);

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

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.45 CLZ

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

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

32-bit 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);
integer datasize = if sf == '1' then 64 else 32;
```

Assembler symbols

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

Operation

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

result = CountLeadingZeroBits(operand1);
X[d] = result<datasize-1:0>;
```

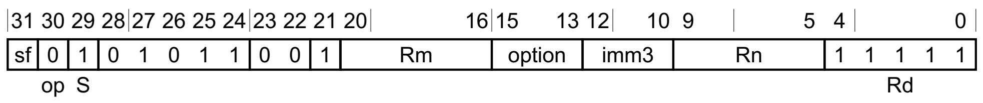
Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.46 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 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.

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.47 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 for this instruction.

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

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 "shift" field. It can have the following values:

`LSL #0`	when `shift = 00`
`LSL #12`	when `shift = 01`

The encoding `shift = 1x` is reserved.

Operation

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

C6.2.48 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 for this instruction.

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

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.

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

The description of ADDS (shifted register) gives the operational pseudocode for this instruction.

C6.2.49 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 for this instruction.

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

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.

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

`<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

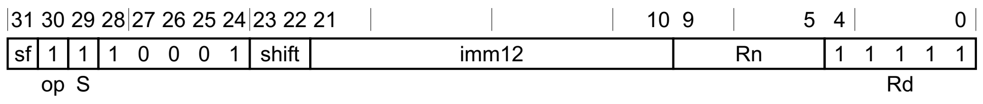
The description of SUBS (extended register) gives the operational pseudocode for this instruction.

C6.2.50 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 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 "shift" field. It can have the following values:

`LSL #0`	when `shift = 00`
`LSL #12`	when `shift = 01`

The encoding `shift = 1x` is reserved.

Operation

The description of SUBS (immediate) gives the operational pseudocode for this instruction.

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.51 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 for this instruction.

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

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.

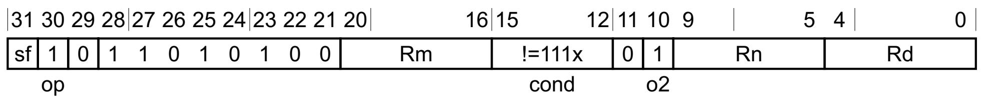
Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.52 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 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.

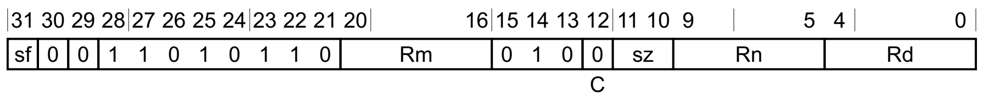
C6.2.53 CRC32B, CRC32H, CRC32W, CRC32X

`CRC32` checksum performs a cyclic redundancy check (CRC) calculation on a value held in a general-purpose register. It takes an input CRC value in the first source operand, performs a CRC on the input value in the second source operand, and returns the output CRC value. The second source operand can be 8, 16, 32, or 64 bits. To align with common usage, the bit order of the values is reversed as part of the operation, and the polynomial `0x04C11DB7` is used for the CRC calculation.

In ARMv8-A, this is an OPTIONAL instruction, and in ARMv8.1 it is mandatory for all implementations to implement it.

Note

ID_AA64ISAR0_EL1.CRC32 indicates whether this instruction is supported.



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

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sf == '1' && sz != '11' then UnallocatedEncoding();
if sf == '0' && sz == '11' then UnallocatedEncoding();
integer size = 8 << UInt(sz);
```

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.

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

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

bits(32) acc = X[n];    // accumulator
bits(size) val = X[m];    // 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] = BitReverse(Poly32Mod2(tempacc EOR tempval, poly));
```

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.54 CRC32CB, CRC32CH, CRC32CW, CRC32CX

`CRC32` checksum performs a cyclic redundancy check (CRC) calculation on a value held in a general-purpose register. It takes an input CRC value in the first source operand, performs a CRC on the input value in the second source operand, and returns the output CRC value. The second source operand can be 8, 16, 32, or 64 bits. To align with common usage, the bit order of the values is reversed as part of the operation, and the polynomial `0x1EDC6F41` is used for the CRC calculation.

In ARMv8-A, this is an OPTIONAL instruction, and in ARMv8.1 it is mandatory for all implementations to implement it.

Note

ID_AA64ISAR0_EL1.CRC32 indicates whether this instruction is supported.

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

CRC32CB 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

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sf == '1' && sz != '11' then UnallocatedEncoding();
if sf == '0' && sz == '11' then UnallocatedEncoding();
integer size = 8 << UInt(sz);
```

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.

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

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

bits(32) acc = X[n];    // accumulator
bits(size) val = X[m];    // 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] = BitReverse(Poly32Mod2(tempacc EOR tempval, poly));
```

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.55 CSEL

Conditional Select returns, in the destination register, the value of the first source register if the condition is TRUE, and otherwise returns the value of the second source register.

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

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);
integer datasize = if sf == '1' then 64 else 32;
```

Assembler symbols

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

Operation

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

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

X[d] = result;
```

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.56 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 for this instruction.

31	30	29	28 27 26 25 24 23 22 21	20 16	15 12	11	10	9 5	4 0
sf	0	0	1 1 0 1 0 1 0 0	1 1 1 1 1	!=111x	0	1	1 1 1 1 1	Rd
	op			Rm	cond		o2	Rn	

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.

C6.2.57 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 for this instruction.

31	30	29	28 27 26 25 24 23 22 21	20 … 16	15 … 12	11	10	9 … 5	4 … 0
sf	1	0	1 1 0 1 0 1 0 0	1 1 1 1 1	!=111x	0	0	1 1 1 1 1	Rd
	op			Rm	cond		o2	Rn	

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

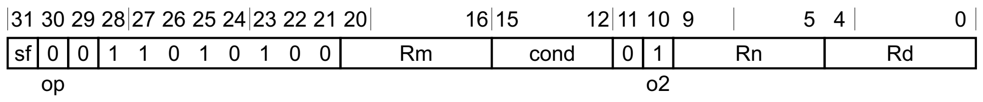
The description of CSINV gives the operational pseudocode for this instruction.

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.58 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);
integer datasize = if sf == '1' then 64 else 32;
```

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

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

X[d] = result;
```

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.59 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.

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

32-bit 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);
integer datasize = if sf == '1' then 64 else 32;
```

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

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

X[d] = result;
```

C6.2.60 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.

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

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);
integer datasize = if sf == '1' then 64 else 32;
```

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

if ConditionHolds(cond) then
```

```
    result = operand1;
else
    result = NOT(operand2);
    result = result + 1;

X[d] = result;
```

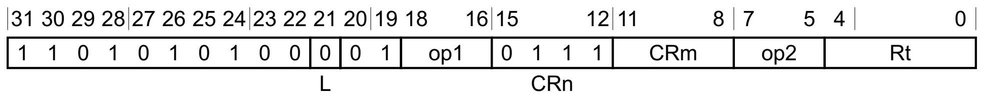
Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.61 DC

Data Cache operation. For more information, see A64 system instructions for cache maintenance.

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 for this instruction.



System variant

`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 ARMv8.2-DCPoP is implemented, the following value is also valid:

CVAP	when op1 = 011, CRm = 1100, 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 general-purpose source register, encoded in the "Rt" field.

Operation

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

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.62 DCPS1

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

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

The target exception level of a DCPS1 instruction is:

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

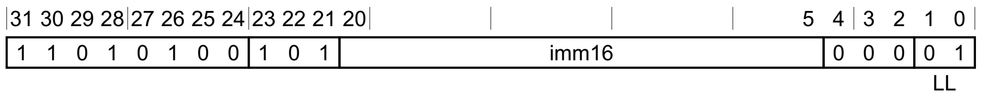
When the target Exception level of a DCPS1 instruction is ELx, on executing this instruction:

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

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

This instruction is always UNDEFINED in Non-debug state.

For more information on the operation of the DCPSn instructions, see *DCPS<n>* on page H2-5461.



System variant

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

Decode for this encoding

```
if !Halted() then AArch64.UndefinedFault();
```

Assembler symbols

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

Operation

```
DCPSInstruction(LL);
```

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.63 DCPS2

Debug Change PE State to EL2, when executed in Debug state:

- If executed at EL0 or EL1 changes the current Exception level and SP to EL2 using SP_EL2.
- Otherwise, if executed at ELx, selects SP_ELx.

The target exception level of a DCPS2 instruction is:

- EL2 if the instruction is executed at an exception level that is not EL3.
- EL3 if the instruction is executed at EL3.

When the target Exception level of a DCPS2 instruction is ELx, on executing this instruction:

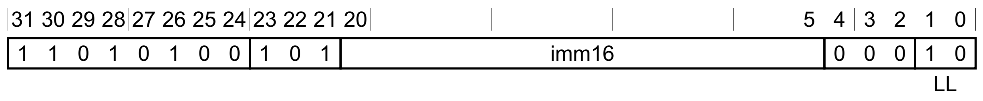
- ELR_ELx becomes UNKNOWN.
- SPSR_ELx becomes UNKNOWN.
- ESR_ELx becomes UNKNOWN.
- DLR_EL0 and DSPSR_EL0 become UNKNOWN.
- The endianness is set according to SCTLR_ELx.EE.

This instruction is UNDEFINED at the following exception levels:

- All exception levels if EL2 is not implemented.
- At EL0 and EL1 in Secure state if EL2 is implemented.

This instruction is always UNDEFINED in Non-debug state.

For more information on the operation of the DCPSn instructions, see *DCPS<n>* on page H2-5461.



System variant

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

Decode for this encoding

```
if !Halted() then AArch64.UndefinedFault();
```

Assembler symbols

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

Operation

```
DCPSInstruction(LL);
```

C6.2.64 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:

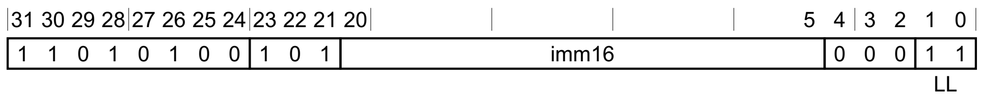
- ELR_EL3 becomes UNKNOWN.
- SPSR_EL3 becomes UNKNOWN.
- ESR_EL3 becomes UNKNOWN.
- DLR_EL0 and DSPSR_EL0 become UNKNOWN.
- The endianness is set according to SCTLR_EL3.EE.

This instruction is UNDEFINED at all exception levels if either:

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

This instruction is always UNDEFINED in Non-debug state.

For more information on the operation of the DCPSn instructions, see *DCPS<n>* on page H2-5461.



System variant

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

Decode for this encoding

```
if !Halted() then AArch64.UndefinedFault();
```

Assembler symbols

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

Operation

```
DCPSInstruction(LL);
```

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.65 DMB

Data Memory Barrier is a memory barrier that ensures the ordering of observations of memory accesses, see *Data Memory Barrier (DMB)* on page B2-92.

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

System variant

```
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 '01' types = MBReqTypes_Reads;
    when '10' types = MBReqTypes_Writes;
    when '11' types = MBReqTypes_All;
    otherwise
        types = MBReqTypes_All;
        domain = MBReqDomain_FullSystem;
```

Assembler symbols

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

`SY`	Full system is the required shareability domain, reads and writes are the required access types, both before and after the barrier instruction. This option is referred to as the full system barrier. Encoded as CRm = `0b1111`.
`ST`	Full system is the required shareability domain, writes are the required access type, both before and after the barrier instruction. Encoded as CRm = `0b1110`.
`LD`	Full system is the required shareability domain, reads are the required access type before the barrier instruction, and reads and writes are the required access types after the barrier instruction. Encoded as CRm = `0b1101`.
`ISH`	Inner Shareable is the required shareability domain, reads and writes are the required access types, both before and after the barrier instruction. Encoded as CRm = `0b1011`.
`ISHST`	Inner Shareable is the required shareability domain, writes are the required access type, both before and after the barrier instruction. Encoded as CRm = `0b1010`.
`ISHLD`	Inner Shareable is the required shareability domain, reads are the required access type before the barrier instruction, and reads and writes are the required access types after the barrier instruction. Encoded as CRm = `0b1001`.
`NSH`	Non-shareable is the required shareability domain, reads and writes are the required access, both before and after the barrier instruction. Encoded as CRm = `0b0111`.
`NSHST`	Non-shareable is the required shareability domain, writes are the required access type before the barrier instruction, and reads and writes are the required type after the barrier instruction. Encoded as CRm = `0b0110`.

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

`NSHLD`	Non-shareable is the required shareability domain, reads are the required access type before the barrier instruction, and reads and writes are the required access types after the barrier instruction. Encoded as CRm = `0b0101`.
`OSH`	Outer Shareable is the required shareability domain, reads and writes are the required access types, both before and after the barrier instruction. Encoded as CRm = `0b0011`.
`OSHST`	Outer Shareable is the required shareability domain, writes are the required access type, both before and after the barrier instruction. Encoded as CRm = `0b0010`.
`OSHLD`	Outer Shareable is the required shareability domain, reads are the required access type before the barrier instruction, and reads and writes are the required access types after the barrier instruction. Encoded as CRm = `0b0001`.

All other encodings of `CRm` that are not listed above are reserved, and can be encoded using the `#<imm>` syntax. It is IMPLEMENTATION DEFINED whether options other than `SY` are implemented. All unsupported and reserved options must execute as a full system barrier operation, but software must not rely on this behavior. For more information on whether an access is before or after a barrier instruction, see *Data Memory Barrier (DMB)* on page B2-92 or see *Data Synchronization Barrier (DSB)* on page B2-93.

`<imm>` Is a 4-bit unsigned immediate, in the range 0 to 15, encoded in the "CRm" field.

Operation

```
DataMemoryBarrier(domain, types);
```

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.66 DRPS

Debug restore process state

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

System variant

```
DRPS
```

Decode for this encoding

```
if !Halted() || PSTATE.EL == EL0 then UnallocatedEncoding();
```

Operation

```
DRPSInstruction();
```

C6.2.67 DSB

Data Synchronization Barrier is a memory barrier that ensures the completion of memory accesses, see *Data Synchronization Barrier (DSB)* on page B2-93.

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

System variant

```
DSB <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 '01' types = MBReqTypes_Reads;
    when '10' types = MBReqTypes_Writes;
    when '11' types = MBReqTypes_All;
    otherwise
        types = MBReqTypes_All;
        domain = MBReqDomain_FullSystem;
```

Assembler symbols

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

SY	Full system is the required shareability domain, reads and writes are the required access types, both before and after the barrier instruction. This option is referred to as the full system barrier. Encoded as CRm = 0b1111.
ST	Full system is the required shareability domain, writes are the required access type, both before and after the barrier instruction. Encoded as CRm = 0b1110.
LD	Full system is the required shareability domain, reads are the required access type before the barrier instruction, and reads and writes are the required access types after the barrier instruction. Encoded as CRm = 0b1101.
ISH	Inner Shareable is the required shareability domain, reads and writes are the required access types, both before and after the barrier instruction. Encoded as CRm = 0b1011.
ISHST	Inner Shareable is the required shareability domain, writes are the required access type, both before and after the barrier instruction. Encoded as CRm = 0b1010.
ISHLD	Inner Shareable is the required shareability domain, reads are the required access type before the barrier instruction, and reads and writes are the required access types after the barrier instruction. Encoded as CRm = 0b1001.
NSH	Non-shareable is the required shareability domain, reads and writes are the required access, both before and after the barrier instruction. Encoded as CRm = 0b0111.
NSHST	Non-shareable is the required shareability domain, writes are the required access type before the barrier instruction, and reads and writes are the required type after the barrier instruction. Encoded as CRm = 0b0110.

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

`NSHLD`	Non-shareable is the required shareability domain, reads are the required access type before the barrier instruction, and reads and writes are the required access types after the barrier instruction. Encoded as CRm = `0b0101`.
`OSH`	Outer Shareable is the required shareability domain, reads and writes are the required access types, both before and after the barrier instruction. Encoded as CRm = `0b0011`.
`OSHST`	Outer Shareable is the required shareability domain, writes are the required access type, both before and after the barrier instruction. Encoded as CRm = `0b0010`.
`OSHLD`	Outer Shareable is the required shareability domain, reads are the required access type before the barrier instruction, and reads and writes are the required access types after the barrier instruction. Encoded as CRm = `0b0001`.

All other encodings of `CRm` that are not listed above are reserved, and can be encoded using the `#<imm>` syntax. It is IMPLEMENTATION DEFINED whether options other than `SY` are implemented. All unsupported and reserved options must execute as a full system barrier operation, but software must not rely on this behavior. For more information on whether an access is before or after a barrier instruction, see *Data Memory Barrier (DMB)* on page B2-92 or see *Data Synchronization Barrier (DSB)* on page B2-93.

`<imm>` Is a 4-bit unsigned immediate, in the range 0 to 15, encoded in the "CRm" field.

Operation

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

C6.2.68 EON (shifted register)

Bitwise Exclusive OR NOT (shifted register) performs a bitwise Exclusive OR NOT of a register value and an optionally-shifted register value, and writes the result to the destination register.

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

32-bit 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);
integer datasize = if sf == '1' then 64 else 32;
if sf == '0' && imm6<5> == '1' then ReservedValue();

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

Assembler symbols

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

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

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

operand2 = NOT(operand2);

result = operand1 EOR operand2;

X[d] = result;
```

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.69 EOR (immediate)

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

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

32-bit 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);
integer datasize = if sf == '1' then 64 else 32;
bits(datasize) imm;
if sf == '0' && N != '0' then ReservedValue();
(imm, -) = DecodeBitMasks(N, imms, immr, TRUE);
```

Assembler symbols

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

Operation

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

result = operand1 EOR imm;

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

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.70 EOR (shifted register)

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

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

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);
integer datasize = if sf == '1' then 64 else 32;
if sf == '0' && imm6<5> == '1' then ReservedValue();

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

Assembler symbols

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

`<shift>` Is the optional shift 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];
bits(datasize) operand2 = ShiftReg(m, shift_type, shift_amount);

result = operand1 EOR operand2;

X[d] = result;
```

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.71 ERET

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

The PE checks the SPSR for the current Exception level for an illegal return event. See *Illegal return events from AArch64 state* on page D1-1816.

`ERET` is UNDEFINED at EL0.

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

System variant

```
ERET
```

Decode for this encoding

```
if PSTATE.EL == EL0 then UnallocatedEncoding();
```

Operation

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

C6.2.72 ESB

Error Synchronization Barrier is a synchronization barrier instruction to barrier between errors. This instruction can be used at all Exception levels and in Debug state. This instruction might update DISR_EL1 and VDISR_EL2.

In Debug state, this instruction behaves as if SError interrupts are masked at all Exception levels. See Error Synchronization Barrier in the *ARM® Reliability, Availability, and Serviceability (RAS) Specification, ARMv8, for ARMv8-A Architecture Profile.*

ARMv8.2

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

System variant

```
ESB
```

Decode for this encoding

```
SystemHintOp op;

op = if HaveRASExt() then SystemHintOp_ESB else SystemHintOp_NOP;
```

Operation

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

    when SystemHintOp_WFE
        if IsEventRegisterSet() then
            ClearEventRegister();
        else
            if PSTATE.EL == EL0 then
                // Check for traps described by the OS which may be EL1 or EL2.
                AArch64.CheckForWFxTrap(EL1, TRUE);
            if HaveEL(EL2) && !IsSecure() && PSTATE.EL IN {EL0, EL1} && !IsInHost() then
                // Check for traps described by the Hypervisor.
                AArch64.CheckForWFxTrap(EL2, TRUE);
            if HaveEL(EL3) && PSTATE.EL != EL3 then
                // Check for traps described by the Secure Monitor.
                AArch64.CheckForWFxTrap(EL3, TRUE);
            WaitForEvent();

    when SystemHintOp_WFI
        if !InterruptPending() then
            if PSTATE.EL == EL0 then
                // Check for traps described by the OS which may be EL1 or EL2.
                AArch64.CheckForWFxTrap(EL1, FALSE);
            if HaveEL(EL2) && !IsSecure() && PSTATE.EL IN {EL0, EL1} && !IsInHost() then
                // Check for traps described by the Hypervisor.
                AArch64.CheckForWFxTrap(EL2, FALSE);
            if HaveEL(EL3) && PSTATE.EL != EL3 then
                // Check for traps described by the Secure Monitor.
                AArch64.CheckForWFxTrap(EL3, FALSE);
            WaitForInterrupt();

    when SystemHintOp_SEV
        SendEvent();
```

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

```
when SystemHintOp_SEVL
    SendEventLocal();

when SystemHintOp_ESB
    ErrorSynchronizationBarrier(MBReqDomain_FullSystem, MBReqTypes_All);
    AArch64.ESBOperation();
    if HaveEL(EL2) && !IsSecure() && PSTATE.EL IN {EL0, EL1} then AArch64.vESBOperation();
    TakeUnmaskedSErrorInterrupts();

when SystemHintOp_PSB
    ProfilingSynchronizationBarrier();

otherwise    // do nothing
```

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.73 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.

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

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

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

if N != sf then UnallocatedEncoding();
if sf == '0' && imms<5> == '1' then ReservedValue();
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.

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
bits(datasize) result;
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = X[m];
bits(2*datasize) concat = operand1:operand2;

result = concat<lsb+datasize-1:lsb>;

X[d] = result;
```

C6.2.74 HINT

Hint instruction is for the instruction set space that is reserved for architectural hint instructions.

Some encodings described here are unallocated 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.

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

Hints 6 and 7 variant

Applies when `CRm == 0000 && op2 == 11x`.

```
HINT #<imm>
```

Hints 8 to 15, and 24 to 127 variant

Applies when `CRm != 00x0`.

```
HINT #<imm>
```

Hints 17 to 23 variant

Applies when `CRm == 0010 && op2 != 00x`.

```
HINT #<imm>
```

Assembler symbols

`<imm>` Is a 7-bit unsigned immediate, in the range 0 to 127, excluding the allocated encodings described below, encoded in "CRm:op2".

The following encodings of "CRm:op2" are allocated:

`0000000`	`NOP`
`0000001`	`YIELD`
`0000010`	`WFE`
`0000011`	`WFI`
`0000100`	`SEV`
`0000101`	`SEVL`

——— 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.

———————————

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.75 HLT

Halt instruction generates a Halt Instruction debug event.

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

System variant

```
HLT #<imm>
```

Decode for this encoding

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

Assembler symbols

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

Operation

```
Halt(DebugHalt_HaltInstruction);
```

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.76 HVC

Hypervisor Call causes an exception to EL2. Non-secure software executing at EL1 can use this instruction to call the hypervisor to request a service.

The `HVC` instruction is UNDEFINED:

- At EL0, and Secure EL1.
- When SCR_EL3.HCE is set to 0.

On executing an `HVC` instruction, the PE records the exception as a Hypervisor Call exception in ESR_ELx, using the EC value `0x16`, and the value of the immediate argument.

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

System variant

```
HVC #<imm>
```

Decode for this encoding

```
bits(16) imm = imm16;
```

Assembler symbols

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

Operation

```
if !HaveEL(EL2) || PSTATE.EL == EL0 || (PSTATE.EL == EL1 && IsSecure()) then
    UnallocatedEncoding();

hvc_enable = if HaveEL(EL3) then SCR_EL3.HCE else NOT(HCR_EL2.HCD);
if hvc_enable == '0' then
    AArch64.UndefinedFault();
else
    AArch64.CallHypervisor(imm);
```

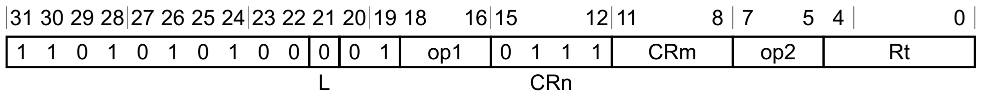
Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.77 IC

Instruction Cache operation. For more information, see A64 system instructions for cache maintenance.

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 for this instruction.



System variant

`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.

C6.2.78 ISB

Instruction Synchronization Barrier flushes the pipeline in the PE and is a context synchronization event. For more information, see *Instruction Synchronization Barrier (ISB)* on page B2-92.

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

System variant

```
ISB {<option>|#<imm>}
```

Decode for this encoding

```
// Empty.
```

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();
```

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.79 LDADDB, LDADDAB, LDADDALB, LDADDLB

Atomic add on byte in memory atomically loads an 8-bit byte from memory, adds the value held in a register to it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not `WZR`, `LDADDAB` and `LDADDALB` load from memory with acquire semantics.
- `LDADDLB` and `LDADDALB` store to memory with release semantics.
- `LDADDB` has no memory ordering requirements.

For more information about memory ordering semantics see *Load-Acquire, Store-Release* on page B2-94.

For information about memory accesses see *Load/Store addressing modes* on page C1-143.

ARMv8.1

31 30	29 28 27	26	25 24	23	22	21	20 16	15	14 12	11 10	9 5	4 0
0 0	1 1 1	0	0 0	A	R	1	Rs	0	0 0 0	0 0	Rn	Rt
size		V						o3	opc			

Acquire variant

Applies when `A == 1 && R == 0`.

```
LDADDAB <Ws>, <Wt>, [<Xn|SP>]
```

Acquire and release variant

Applies when `A == 1 && R == 1`.

```
LDADDALB <Ws>, <Wt>, [<Xn|SP>]
```

No memory ordering variant

Applies when `A == 0 && R == 0 && Rt != 11111`.

```
LDADDB <Ws>, <Wt>, [<Xn|SP>]
```

Release variant

Applies when `A == 0 && R == 1 && Rt != 11111`.

```
LDADDLB <Ws>, <Wt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !HaveAtomicExt() then UnallocatedEncoding();
integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
```

Assembler symbols

`<Ws>`	Is the 32-bit name of the general-purpose register 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;
bits(8) result;

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, 1, ldacctype];

result = data + value;
// All observers in the shareability domain observe the
// following load and store atomically.
Mem[address, 1, stacctype] = result;

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

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.80 LDADDH, LDADDAH, LDADDALH, LDADDLH

Atomic add on halfword in memory atomically loads a 16-bit halfword from memory, adds the value held in a register to it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not `WZR`, `LDADDAH` and `LDADDALH` load from memory with acquire semantics.
- `LDADDLH` and `LDADDALH` store to memory with release semantics.
- `LDADDH` has no memory ordering requirements.

For more information about memory ordering semantics see *Load-Acquire, Store-Release* on page B2-94.

For information about memory accesses see *Load/Store addressing modes* on page C1-143.

ARMv8.1

31 30	29 28 27	26	25 24	23	22	21	20 16	15	14 12	11 10	9 5	4 0
0 1	1 1 1	0	0 0	A	R	1	Rs	0	0 0 0	0 0	Rn	Rt
size		V						o3	opc			

Acquire variant

Applies when `A == 1 && R == 0`.

```
LDADDAH <Ws>, <Wt>, [<Xn|SP>]
```

Acquire and release variant

Applies when `A == 1 && R == 1`.

```
LDADDALH <Ws>, <Wt>, [<Xn|SP>]
```

No memory ordering variant

Applies when `A == 0 && R == 0 && Rt != 11111`.

```
LDADDH <Ws>, <Wt>, [<Xn|SP>]
```

Release variant

Applies when `A == 0 && R == 1 && Rt != 11111`.

```
LDADDLH <Ws>, <Wt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !HaveAtomicExt() then UnallocatedEncoding();
integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
```

Assembler symbols

`<Ws>`	Is the 32-bit name of the general-purpose register 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;
bits(16) result;

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, 2, ldacctype];

result = data + value;
// All observers in the shareability domain observe the
// following load and store atomically.
Mem[address, 2, stacctype] = result;

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

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.81 LDADD, LDADDA, LDADDAL, LDADDL

Atomic add on word or doubleword in memory atomically loads a 32-bit word or 64-bit doubleword from memory, adds the value held in a register to it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not one of `WZR` or `XZR`, `LDADDA` and `LDADDAL` load from memory with acquire semantics.
- `LDADDL` and `LDADDAL` store to memory with release semantics.
- `LDADD` has no memory ordering requirements.

For more information about memory ordering semantics see *Load-Acquire, Store-Release* on page B2-94.

For information about memory accesses see *Load/Store addressing modes* on page C1-143.

ARMv8.1

31 30	29 28 27	26	25 24	23	22	21	20 16	15	14 12	11 10	9 5	4 0
1 x	1 1 1	0	0 0	A	R	1	Rs	0	0 0 0	0 0	Rn	Rt
size		V						o3	opc			

32-bit, acquire variant

Applies when `size == 10 && A == 1 && R == 0`.

```
LDADDA <Ws>, <Wt>, [<Xn|SP>]
```

32-bit, acquire and release variant

Applies when `size == 10 && A == 1 && R == 1`.

```
LDADDAL <Ws>, <Wt>, [<Xn|SP>]
```

32-bit, no memory ordering variant

Applies when `size == 10 && A == 0 && R == 0 && Rt != 11111`.

```
LDADD <Ws>, <Wt>, [<Xn|SP>]
```

32-bit, release variant

Applies when `size == 10 && A == 0 && R == 1 && Rt != 11111`.

```
LDADDL <Ws>, <Wt>, [<Xn|SP>]
```

64-bit, acquire variant

Applies when `size == 11 && A == 1 && R == 0`.

```
LDADDA <Xs>, <Xt>, [<Xn|SP>]
```

64-bit, acquire and release variant

Applies when `size == 11 && A == 1 && R == 1`.

```
LDADDAL <Xs>, <Xt>, [<Xn|SP>]
```

64-bit, no memory ordering variant

Applies when `size == 11 && A == 0 && R == 0 && Rt != 11111`.

```
LDADD <Xs>, <Xt>, [<Xn|SP>]
```

64-bit, release variant

Applies when `size == 11 && A == 0 && R == 1 && Rt != 11111`.

```
LDADDL <Xs>, <Xt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !HaveAtomicExt() then UnallocatedEncoding();
integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
```

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;
bits(datasize) result;

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, datasize DIV 8, ldacctype];

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

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

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.82 LDAR

Load-Acquire Register derives an address from a base register value, loads a 32-bit word or 64-bit doubleword from memory, and writes it to a register. The instruction also has memory ordering semantics as described in *Load-Acquire, Store-Release* on page B2-94. For information about memory accesses, see *Load/Store addressing modes* on page C1-143.

31	30	29	28	27	26	25	24	23	22	21	20 16	15	14 10	9 5	4 0
1	x	0	0	1	0	0	0	1	1	0	(1) (1) (1) (1) (1)	1	(1) (1) (1) (1) (1)	Rn	Rt
size									L		Rs	o0	Rt2		

32-bit 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);

integer elsize = 8 << UInt(size);
integer regsize = if elsize == 64 then 64 else 32;
```

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;

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

data = Mem[address, dbytes, AccType_ORDERED];
X[t] = ZeroExtend(data, regsize);
```

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.83 LDARB

Load-Acquire Register Byte derives an address from a base register value, loads a byte from memory, zero-extends it and writes it to a register. The instruction also has memory ordering semantics as described in *Load-Acquire, Store-Release* on page B2-94. For information about memory accesses, see *Load/Store addressing modes* on page C1-143.

31	30	29	28	27	26	25	24	23	22	21	20 … 16	15	14 … 10	9 … 5	4 … 0
0	0	0	0	1	0	0	0	1	1	0	(1) (1) (1) (1) (1)	1	(1) (1) (1) (1) (1)	Rn	Rt
size									L		Rs	o0	Rt2		

No offset variant

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

Decode for this encoding

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

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;

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

data = Mem[address, 1, AccType_ORDERED];
X[t] = ZeroExtend(data, 32);
```

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.84 LDARH

Load-Acquire Register Halfword derives an address from a base register value, loads a halfword from memory, zero-extends it, and writes it to a register. The instruction also has memory ordering semantics as described in *Load-Acquire, Store-Release* on page B2-94. For information about memory accesses, see *Load/Store addressing modes* on page C1-143.

31 30	29 28 27 26 25 24	23	22	21	20 16	15	14 10	9 5	4 0
0 1	0 0 1 0 0 0	1	1	0	(1) (1) (1) (1) (1)	1	(1) (1) (1) (1) (1)	Rn	Rt
size			L		Rs	o0	Rt2		

No offset variant

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

Decode for this encoding

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

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;

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

data = Mem[address, 2, AccType_ORDERED];
X[t] = ZeroExtend(data, 32);
```

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.85 LDAXP

Load-Acquire Exclusive Pair of Registers derives an address from a base register value, loads two 32-bit words or two 64-bit doublewords from memory, and writes them to two registers. A 32-bit pair requires the address to be doubleword aligned and is single-copy atomic at doubleword granularity. A 64-bit pair requires the address to be quadword aligned and is single-copy atomic for each doubleword at doubleword granularity. The PE marks the physical address being accessed as an exclusive access. This exclusive access mark is checked by Store Exclusive instructions. See *Synchronization and semaphores* on page B2-121. The instruction also has memory ordering semantics as described in *Load-Acquire, Store-Release* on page B2-94. For information about memory accesses see *Load/Store addressing modes* on page C1-143.

31	30	29 28 27 26 25 24	23	22	21	20 16	15	14 10	9 5	4 0
1	sz	0 0 1 0 0 0	0	1	1	(1) (1) (1) (1) (1)	1	Rt2	Rn	Rt
				L		Rs	o0			

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);

integer elsize = 32 << UInt(sz);
integer datasize = elsize * 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 *LDAXP* on page K1-6124.

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.

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
bits(64) address;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;

if t == t2 then
    Constraint c = ConstrainUnpredictable();
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN rt_unknown = TRUE;    // result is UNKNOWN
        when Constraint_UNDEF   UnallocatedEncoding();
        when Constraint_NOP     EndOfInstruction();

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

// Tell the Exclusive Monitors to record a sequence of one or more atomic
// memory reads from virtual address range [address, address+dbytes-1].
// The Exclusive Monitor will only be set if all the reads are from the
// same dbytes-aligned physical address, to allow for the possibility of
// an atomicity break if the translation is changed between reads.
AArch64.SetExclusiveMonitors(address, dbytes);

if rt_unknown then
    // ConstrainedUNPREDICTABLE case
    X[t] = bits(datasize) UNKNOWN;
elsif elsize == 32 then
    // 32-bit load exclusive pair (atomic)
    data = Mem[address, dbytes, AccType_ORDERED];
    if BigEndian() then
        X[t] = data<datasize-1:elsize>;
        X[t2] = data<elsize-1:0>;
    else
        X[t] = data<elsize-1:0>;
        X[t2] = data<datasize-1:elsize>;
else // elsize == 64
    // 64-bit load exclusive pair (not atomic),
    // but must be 128-bit aligned
    if address != Align(address, dbytes) then
        AArch64.Abort(address, AArch64.AlignmentFault(AccType_ORDERED, FALSE, FALSE));
    X[t] = Mem[address, 8, AccType_ORDERED];
    X[t2] = Mem[address+8, 8, AccType_ORDERED];
```

C6.2.86 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* on page B2-121. The instruction also has memory ordering semantics as described in *Load-Acquire, Store-Release* on page B2-94. For information about memory accesses see *Load/Store addressing modes* on page C1-143.

31 30	29 28 27 26 25 24	23	22	21	20 … 16	15	14 … 10	9 … 5	4 … 0
1 x	0 0 1 0 0 0	0	1	0	(1) (1) (1) (1) (1)	1	(1) (1) (1) (1) (1)	Rn	Rt
size			L		Rs	o0	Rt2		

32-bit 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);

integer elsize = 8 << UInt(size);
integer regsize = if elsize == 64 then 64 else 32;
```

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;

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

// Tell the Exclusive Monitors to record a sequence of one or more atomic
// memory reads from virtual address range [address, address+dbytes-1].
// The Exclusive Monitor will only be set if all the reads are from the
// same dbytes-aligned physical address, to allow for the possibility of
// an atomicity break if the translation is changed between reads.
AArch64.SetExclusiveMonitors(address, dbytes);
```

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

```
            data = Mem[address, dbytes, AccType_ORDERED];
            X[t] = ZeroExtend(data, regsize);
```

C6.2.87 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* on page B2-121. The instruction also has memory ordering semantics as described in *Load-Acquire, Store-Release* on page B2-94. For information about memory accesses see *Load/Store addressing modes* on page C1-143.

31 30	29 28 27 26 25 24	23	22	21	20 16	15	14 10	9 5	4 0
0 0	0 0 1 0 0 0	0	1	0	(1) (1) (1) (1) (1)	1	(1) (1) (1) (1) (1)	Rn	Rt
size			L		Rs	o0	Rt2		

No offset variant

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

Decode for this encoding

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

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;

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

// Tell the Exclusive Monitors to record a sequence of one or more atomic
// memory reads from virtual address range [address, address+dbytes-1].
// The Exclusive Monitor will only be set if all the reads are from the
// same dbytes-aligned physical address, to allow for the possibility of
// an atomicity break if the translation is changed between reads.
AArch64.SetExclusiveMonitors(address, 1);

data = Mem[address, 1, AccType_ORDERED];
X[t] = ZeroExtend(data, 32);
```

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.88 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* on page B2-121. The instruction also has memory ordering semantics as described in *Load-Acquire, Store-Release* on page B2-94. For information about memory accesses see *Load/Store addressing modes* on page C1-143.

31 30	29 28 27 26 25 24	23	22	21	20 ... 16	15	14 ... 10	9 ... 5	4 ... 0
0 1	0 0 1 0 0 0	0	1	0	(1) (1) (1) (1) (1)	1	(1) (1) (1) (1) (1)	Rn	Rt
size			L		Rs	o0	Rt2		

No offset variant

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

Decode for this encoding

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

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;

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

// Tell the Exclusive Monitors to record a sequence of one or more atomic
// memory reads from virtual address range [address, address+dbytes-1].
// The Exclusive Monitor will only be set if all the reads are from the
// same dbytes-aligned physical address, to allow for the possibility of
// an atomicity break if the translation is changed between reads.
AArch64.SetExclusiveMonitors(address, 2);

data = Mem[address, 2, AccType_ORDERED];
X[t] = ZeroExtend(data, 32);
```

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.89 LDCLRB, LDCLRAB, LDCLRALB, LDCLRLB

Atomic bit clear on byte in memory atomically loads an 8-bit byte from memory, performs a bitwise AND with the complement of the value held in a register on it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not `WZR`, `LDCLRAB` and `LDCLRALB` load from memory with acquire semantics.
- `LDCLRLB` and `LDCLRALB` store to memory with release semantics.
- `LDCLRB` has no memory ordering requirements.

For more information about memory ordering semantics see *Load-Acquire, Store-Release* on page B2-94.

For information about memory accesses see *Load/Store addressing modes* on page C1-143.

ARMv8.1

31 30	29 28 27	26	25 24	23	22	21	20 16	15	14 12	11 10	9 5	4 0
0 0	1 1 1	0	0 0	A	R	1	Rs	0	0 0 1	0 0	Rn	Rt
size		V						o3	opc			

Acquire variant

Applies when `A == 1 && R == 0`.

```
LDCLRAB <Ws>, <Wt>, [<Xn|SP>]
```

Acquire and release variant

Applies when `A == 1 && R == 1`.

```
LDCLRALB <Ws>, <Wt>, [<Xn|SP>]
```

No memory ordering variant

Applies when `A == 0 && R == 0 && Rt != 11111`.

```
LDCLRB <Ws>, <Wt>, [<Xn|SP>]
```

Release variant

Applies when `A == 0 && R == 1 && Rt != 11111`.

```
LDCLRLB <Ws>, <Wt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !HaveAtomicExt() then UnallocatedEncoding();
integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
```

Assembler symbols

`<Ws>`	Is the 32-bit name of the general-purpose register 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.

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

`<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;
bits(8) result;

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, 1, ldacctype];

result = data AND NOT(value);
// All observers in the shareability domain observe the
// following load and store atomically.
Mem[address, 1, stacctype] = result;

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

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.90 LDCLRH, LDCLRAH, LDCLRALH, LDCLRLH

Atomic bit clear on halfword in memory atomically loads a 16-bit halfword from memory, performs a bitwise AND with the complement of the value held in a register on it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not `WZR`, `LDCLRAH` and `LDCLRALH` load from memory with acquire semantics.
- `LDCLRLH` and `LDCLRALH` store to memory with release semantics.
- `LDCLRH` has no memory ordering requirements.

For more information about memory ordering semantics see *Load-Acquire, Store-Release* on page B2-94.

For information about memory accesses see *Load/Store addressing modes* on page C1-143.

ARMv8.1

31 30	29 28 27	26	25 24	23	22	21	20 16	15	14 12	11 10	9 5	4 0
0 1	1 1 1	0	0 0	A	R	1	Rs	0	0 0 1	0 0	Rn	Rt
size		V						o3	opc			

Acquire variant

Applies when `A == 1 && R == 0`.

```
LDCLRAH <Ws>, <Wt>, [<Xn|SP>]
```

Acquire and release variant

Applies when `A == 1 && R == 1`.

```
LDCLRALH <Ws>, <Wt>, [<Xn|SP>]
```

No memory ordering variant

Applies when `A == 0 && R == 0 && Rt != 11111`.

```
LDCLRH <Ws>, <Wt>, [<Xn|SP>]
```

Release variant

Applies when `A == 0 && R == 1 && Rt != 11111`.

```
LDCLRLH <Ws>, <Wt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !HaveAtomicExt() then UnallocatedEncoding();
integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
```

Assembler symbols

`<Ws>`	Is the 32-bit name of the general-purpose register 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.

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

`<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;
bits(16) result;

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, 2, ldacctype];

result = data AND NOT(value);
// All observers in the shareability domain observe the
// following load and store atomically.
Mem[address, 2, stacctype] = result;

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

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.91 LDCLR, LDCLRA, LDCLRAL, LDCLRL

Atomic bit clear on word or doubleword in memory atomically loads a 32-bit word or 64-bit doubleword from memory, performs a bitwise AND with the complement of the value held in a register on it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not one of `WZR` or `XZR`, `LDCLRA` and `LDCLRAL` load from memory with acquire semantics.
- `LDCLRL` and `LDCLRAL` store to memory with release semantics.
- `LDCLR` has no memory ordering requirements.

For more information about memory ordering semantics see *Load-Acquire, Store-Release* on page B2-94.

For information about memory accesses see *Load/Store addressing modes* on page C1-143.

ARMv8.1

31	30	29	28	27	26	25	24	23	22	21	20-16	15	14	13	12	11	10	9-5	4-0
1	x	1	1	1	0	0	0	A	R	1	Rs	0	0	0	1	0	0	Rn	Rt
size					V							o3	opc						

32-bit, acquire variant

Applies when `size == 10 && A == 1 && R == 0`.

```
LDCLRA <Ws>, <Wt>, [<Xn|SP>]
```

32-bit, acquire and release variant

Applies when `size == 10 && A == 1 && R == 1`.

```
LDCLRAL <Ws>, <Wt>, [<Xn|SP>]
```

32-bit, no memory ordering variant

Applies when `size == 10 && A == 0 && R == 0 && Rt != 11111`.

```
LDCLR <Ws>, <Wt>, [<Xn|SP>]
```

32-bit, release variant

Applies when `size == 10 && A == 0 && R == 1 && Rt != 11111`.

```
LDCLRL <Ws>, <Wt>, [<Xn|SP>]
```

64-bit, acquire variant

Applies when `size == 11 && A == 1 && R == 0`.

```
LDCLRA <Xs>, <Xt>, [<Xn|SP>]
```

64-bit, acquire and release variant

Applies when `size == 11 && A == 1 && R == 1`.

```
LDCLRAL <Xs>, <Xt>, [<Xn|SP>]
```

64-bit, no memory ordering variant

Applies when `size == 11 && A == 0 && R == 0 && Rt != 11111`.

```
LDCLR <Xs>, <Xt>, [<Xn|SP>]
```

64-bit, release variant

Applies when `size == 11 && A == 0 && R == 1 && Rt != 11111`.

```
LDCLRL <Xs>, <Xt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !HaveAtomicExt() then UnallocatedEncoding();
integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
```

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;
bits(datasize) result;

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, datasize DIV 8, ldacctype];

result = data AND NOT(value);
// All observers in the shareability domain observe the
// following load and store atomically.
Mem[address, datasize DIV 8, stacctype] = result;

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

C6.2.92 LDEORB, LDEORAB, LDEORALB, LDEORLB

Atomic exclusive OR on byte in memory atomically loads an 8-bit byte from memory, performs an exclusive OR with the value held in a register on it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not `WZR`, `LDEORAB` and `LDEORALB` load from memory with acquire semantics.
- `LDEORLB` and `LDEORALB` store to memory with release semantics.
- `LDEORB` has no memory ordering requirements.

For more information about memory ordering semantics see *Load-Acquire, Store-Release* on page B2-94.

For information about memory accesses see *Load/Store addressing modes* on page C1-143.

ARMv8.1

31 30	29 28 27	26	25 24	23	22	21	20 … 16	15	14 … 12	11 10	9 … 5	4 … 0
0 0	1 1 1	0	0 0	A	R	1	Rs	0	0 1 0	0 0	Rn	Rt
size		V						o3	opc			

Acquire variant

Applies when `A == 1 && R == 0`.

```
LDEORAB <Ws>, <Wt>, [<Xn|SP>]
```

Acquire and release variant

Applies when `A == 1 && R == 1`.

```
LDEORALB <Ws>, <Wt>, [<Xn|SP>]
```

No memory ordering variant

Applies when `A == 0 && R == 0 && Rt != 11111`.

```
LDEORB <Ws>, <Wt>, [<Xn|SP>]
```

Release variant

Applies when `A == 0 && R == 1 && Rt != 11111`.

```
LDEORLB <Ws>, <Wt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !HaveAtomicExt() then UnallocatedEncoding();
integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
```

Assembler symbols

`<Ws>`	Is the 32-bit name of the general-purpose register 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.

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

`<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;
bits(8) result;

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, 1, ldacctype];

result = data EOR value;
// All observers in the shareability domain observe the
// following load and store atomically.
Mem[address, 1, stacctype] = result;

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

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.93 LDEORH, LDEORAH, LDEORALH, LDEORLH

Atomic exclusive OR on halfword in memory atomically loads a 16-bit halfword from memory, performs an exclusive OR with the value held in a register on it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not `WZR`, `LDEORAH` and `LDEORALH` load from memory with acquire semantics.
- `LDEORLH` and `LDEORALH` store to memory with release semantics.
- `LDEORH` has no memory ordering requirements.

For more information about memory ordering semantics see *Load-Acquire, Store-Release* on page B2-94.

For information about memory accesses see *Load/Store addressing modes* on page C1-143.

ARMv8.1

31 30	29 28 27	26	25 24	23	22	21	20 16	15	14 12	11 10	9 5	4 0
0 1	1 1 1	0	0 0	A	R	1	Rs	0	0 1 0	0 0	Rn	Rt
size		V						o3	opc			

Acquire variant

Applies when `A == 1 && R == 0`.

```
LDEORAH <Ws>, <Wt>, [<Xn|SP>]
```

Acquire and release variant

Applies when `A == 1 && R == 1`.

```
LDEORALH <Ws>, <Wt>, [<Xn|SP>]
```

No memory ordering variant

Applies when `A == 0 && R == 0 && Rt != 11111`.

```
LDEORH <Ws>, <Wt>, [<Xn|SP>]
```

Release variant

Applies when `A == 0 && R == 1 && Rt != 11111`.

```
LDEORLH <Ws>, <Wt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !HaveAtomicExt() then UnallocatedEncoding();
integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
```

Assembler symbols

`<Ws>`	Is the 32-bit name of the general-purpose register 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.

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

`<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;
bits(16) result;

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, 2, ldacctype];

result = data EOR value;
// All observers in the shareability domain observe the
// following load and store atomically.
Mem[address, 2, stacctype] = result;

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

C6.2.94 LDEOR, LDEORA, LDEORAL, LDEORL

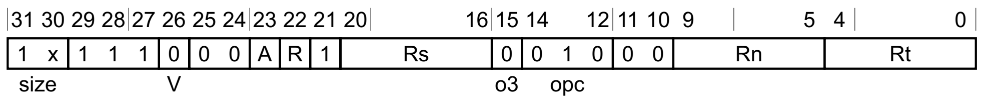
Atomic exclusive OR on word or doubleword in memory atomically loads a 32-bit word or 64-bit doubleword from memory, performs an exclusive OR with the value held in a register on it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not one of `WZR` or `XZR`, `LDEORA` and `LDEORAL` load from memory with acquire semantics.
- `LDEORL` and `LDEORAL` store to memory with release semantics.
- `LDEOR` has no memory ordering requirements.

For more information about memory ordering semantics see *Load-Acquire, Store-Release* on page B2-94.

For information about memory accesses see *Load/Store addressing modes* on page C1-143.

ARMv8.1



32-bit, acquire variant

Applies when `size == 10 && A == 1 && R == 0`.

```
LDEORA <Ws>, <Wt>, [<Xn|SP>]
```

32-bit, acquire and release variant

Applies when `size == 10 && A == 1 && R == 1`.

```
LDEORAL <Ws>, <Wt>, [<Xn|SP>]
```

32-bit, no memory ordering variant

Applies when `size == 10 && A == 0 && R == 0 && Rt != 11111`.

```
LDEOR <Ws>, <Wt>, [<Xn|SP>]
```

32-bit, release variant

Applies when `size == 10 && A == 0 && R == 1 && Rt != 11111`.

```
LDEORL <Ws>, <Wt>, [<Xn|SP>]
```

64-bit, acquire variant

Applies when `size == 11 && A == 1 && R == 0`.

```
LDEORA <Xs>, <Xt>, [<Xn|SP>]
```

64-bit, acquire and release variant

Applies when `size == 11 && A == 1 && R == 1`.

```
LDEORAL <Xs>, <Xt>, [<Xn|SP>]
```

64-bit, no memory ordering variant

Applies when `size == 11 && A == 0 && R == 0 && Rt != 11111`.

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

```
LDEOR <Xs>, <Xt>, [<Xn|SP>]
```

64-bit, release variant

Applies when `size == 11 && A == 0 && R == 1 && Rt != 11111`.

```
LDEORL <Xs>, <Xt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !HaveAtomicExt() then UnallocatedEncoding();
integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
```

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;
bits(datasize) result;

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, datasize DIV 8, ldacctype];

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

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

C6.2.95 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* on page B2-95. For information about memory accesses, see *Load/Store addressing modes* on page C1-143.

ARMv8.1

31	30	29	28	27	26	25	24	23	22	21	20				16	15	14				10	9				5	4				0
0	0	0	0	1	0	0	0	1	1	0	(1)	(1)	(1)	(1)	(1)	0	(1)	(1)	(1)	(1)	(1)	Rn					Rt				
size									L				Rs			o0			Rt2												

No offset variant

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

Decode for this encoding

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

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;

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

data = Mem[address, 1, AccType_LIMITEDORDERED];
X[t] = ZeroExtend(data, 32);
```

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.96 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* on page B2-95. For information about memory accesses, see *Load/Store addressing modes* on page C1-143.

ARMv8.1

31 30	29 28 27 26 25 24	23	22	21	20 16	15	14 10	9 5	4 0
0 1	0 0 1 0 0 0	1	1	0	(1) (1) (1) (1) (1)	0	(1) (1) (1) (1) (1)	Rn	Rt
size			L		Rs	o0	Rt2		

No offset variant

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

Decode for this encoding

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

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;

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

data = Mem[address, 2, AccType_LIMITEDORDERED];
X[t] = ZeroExtend(data, 32);
```

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.97 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* on page B2-95. For information about memory accesses, see *Load/Store addressing modes* on page C1-143.

ARMv8.1

31	30	29	28	27	26	25	24	23	22	21	20 16	15	14 10	9 5	4 0
1	x	0	0	1	0	0	0	1	1	0	(1)(1)(1)(1)(1)	0	(1)(1)(1)(1)(1)	Rn	Rt
size									L		Rs	o0	Rt2		

32-bit 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);

integer elsize = 8 << UInt(size);
integer regsize = if elsize == 64 then 64 else 32;
```

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

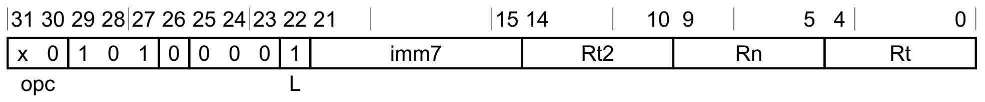
```
bits(64) address;
bits(elsize) data;
constant integer dbytes = elsize DIV 8;

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

data = Mem[address, dbytes, AccType_LIMITEDORDERED];
X[t] = ZeroExtend(data, regsize);
```

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.98 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* on page C1-143. For information about Non-temporal pair instructions, see *Load/Store Non-temporal Pair* on page C3-163.



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* on page K1-6125.

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 UnallocatedEncoding();
```

```
integer scale = 2 + UInt(opc<1>);
integer datasize = 8 << scale;
bits(64) offset = LSL(SignExtend(imm7, 64), scale);
```

Operation

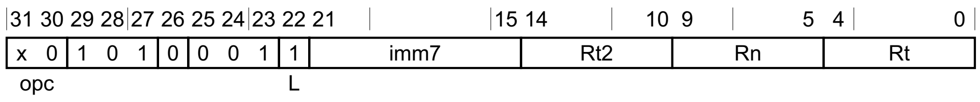
```
bits(64) address;
bits(datasize) data1;
bits(datasize) data2;
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;

if t == t2 then
    Constraint c = ConstrainUnpredictable();
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN rt_unknown = TRUE;    // result is UNKNOWN
        when Constraint_UNDEF   UnallocatedEncoding();
        when Constraint_NOP     EndOfInstruction();

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

address = address + offset;

data1 = Mem[address, dbytes, AccType_STREAM];
data2 = Mem[address+dbytes, dbytes, AccType_STREAM];
if rt_unknown then
    data1 = bits(datasize) UNKNOWN;
    data2 = bits(datasize) UNKNOWN;
X[t] = data1;
X[t2] = data2;
```

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.99 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* on page C1-143.

Post-index



32-bit variant

Applies when `opc == 00`.

```
LDP <Wt1>, <Wt2>, [<Xn|SP>], #<imm>
```

64-bit variant

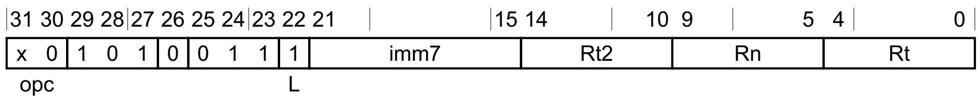
Applies when `opc == 10`.

```
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

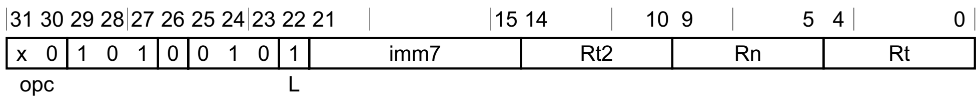
Applies when `opc == 10`.

```
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* on page K1-6125.

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 UnallocatedEncoding();
boolean signed = (opc<0> != '0');
integer scale = 2 + UInt(opc<1>);
integer datasize = 8 << scale;
bits(64) offset = LSL(SignExtend(imm7, 64), scale);
```

Operation for all encodings

```
bits(64) address;
bits(datasize) data1;
bits(datasize) data2;
constant integer dbytes = datasize DIV 8;
```

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

```
boolean rt_unknown = FALSE;
boolean wb_unknown = FALSE;

if wback && (t == n || t2 == n) && n != 31 then
    Constraint c = ConstrainUnpredictable();
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;    // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;    // writeback is UNKNOWN
        when Constraint_UNDEF      UnallocatedEncoding();
        when Constraint_NOP        EndOfInstruction();

if t == t2 then
    Constraint c = ConstrainUnpredictable();
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN rt_unknown = TRUE;    // result is UNKNOWN
        when Constraint_UNDEF   UnallocatedEncoding();
        when Constraint_NOP     EndOfInstruction();

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

if !postindex then
    address = address + offset;

data1 = Mem[address, dbytes, AccType_NORMAL];
data2 = Mem[address+dbytes, dbytes, AccType_NORMAL];
if rt_unknown then
    data1 = bits(datasize) UNKNOWN;
    data2 = bits(datasize) UNKNOWN;
if signed then
    X[t] = SignExtend(data1, 64);
    X[t2] = SignExtend(data2, 64);
else
    X[t] = data1;
    X[t2] = data2;

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```

C6.2.100 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* on page C1-143.

Post-index

31	30	29	28	27	26	25	24	23	22	21 15	14 10	9 5	4 0
0	1	1	0	1	0	0	0	1	1	imm7	Rt2	Rn	Rt
opc									L				

Post-index variant

```
LDPSW <Xt1>, <Xt2>, [<Xn|SP>], #<imm>
```

Decode for this encoding

```
boolean wback = TRUE;
boolean postindex = TRUE;
```

Pre-index

31	30	29	28	27	26	25	24	23	22	21 15	14 10	9 5	4 0
0	1	1	0	1	0	0	1	1	1	imm7	Rt2	Rn	Rt
opc									L				

Pre-index variant

```
LDPSW <Xt1>, <Xt2>, [<Xn|SP>, #<imm>]!
```

Decode for this encoding

```
boolean wback = TRUE;
boolean postindex = FALSE;
```

Signed offset

31	30	29	28	27	26	25	24	23	22	21 15	14 10	9 5	4 0
0	1	1	0	1	0	0	1	0	1	imm7	Rt2	Rn	Rt
opc									L				

Signed offset variant

```
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* on page K1-6126.

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

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);
```

Operation for all encodings

```
bits(64) address;
bits(32) data1;
bits(32) data2;
boolean rt_unknown = FALSE;
boolean wb_unknown = FALSE;

if wback && (t == n || t2 == n) && n != 31 then
    Constraint c = ConstrainUnpredictable();
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;    // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;    // writeback is UNKNOWN
        when Constraint_UNDEF      UnallocatedEncoding();
        when Constraint_NOP        EndOfInstruction();

if t == t2 then
    Constraint c = ConstrainUnpredictable();
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN rt_unknown = TRUE;    // result is UNKNOWN
        when Constraint_UNDEF   UnallocatedEncoding();
        when Constraint_NOP     EndOfInstruction();

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

if !postindex then
    address = address + offset;

data1 = Mem[address, 4, AccType_NORMAL];
data2 = Mem[address+4, 4, AccType_NORMAL];
if rt_unknown then
    data1 = bits(32) UNKNOWN;
    data2 = bits(32) UNKNOWN;
X[t] = SignExtend(data1, 64);
X[t2] = SignExtend(data2, 64);
if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
```

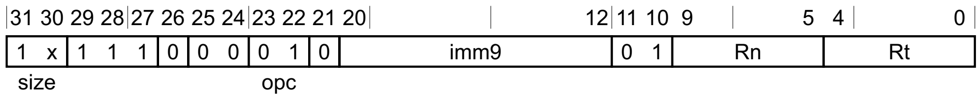
```
elsif postindex then
    address = address + offset;
if n == 31 then
    SP[] = address;
else
    X[n] = address;
```

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.101 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* on page C1-143. 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

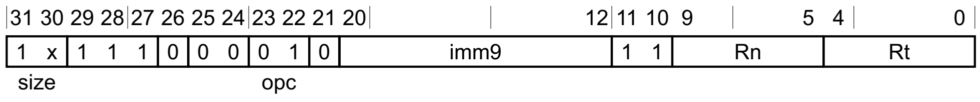
Applies when `size == 11`.

```
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

31	30	29	28	27	26	25	24	23	22	21 10	9 5	4 0
1	x	1	1	1	0	0	1	0	1	imm12	Rn	Rt
size								opc				

32-bit 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)* on page K1-6126.

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;
integer datasize = 8 << scale;
```

Operation for all encodings

```
bits(64) address;
bits(datasize) data;
boolean wb_unknown = FALSE;

if wback && n == t && n != 31 then
    c = ConstrainUnpredictable();
```

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

```
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;    // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;    // writeback is UNKNOWN
        when Constraint_UNDEF      UnallocatedEncoding();
        when Constraint_NOP        EndOfInstruction();

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

if !postindex then
    address = address + offset;

data = Mem[address, datasize DIV 8, AccType_NORMAL];
X[t] = ZeroExtend(data, regsize);

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```

C6.2.102 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* on page C1-143.

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

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 = MemOp_LOAD;
boolean signed = FALSE;
integer size;
bits(64) offset;

case opc of
    when '00'
        size = 4;
    when '01'
        size = 8;
    when '10'
        size = 4;
        signed = TRUE;
    when '11'
        memop = MemOp_PREFETCH;

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

Assembler symbols

`<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 = PC[] + offset;
bits(size*8) data;

case memop of
    when MemOp_LOAD
        data = Mem[address, size, AccType_NORMAL];
        if signed then
```

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

```
            X[t] = SignExtend(data, 64);
        else
            X[t] = data;

    when MemOp_PREFETCH
        Prefetch(address, t<4:0>);
```

C6.2.103 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* on page C1-143.

31 30	29 28 27	26	25 24	23 22	21	20 16	15 13	12	11 10	9 5	4 0
1 x	1 1 1	0	0 0	0 1	1	Rm	option	S	1 0	Rn	Rt
size				opc							

32-bit 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 UnallocatedEncoding();    // sub-word index
ExtendType extend_type = DecodeRegExtend(option);
integer shift = if S == '1' then scale else 0;
```

Assembler symbols

`<Wt>`	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
`<Xt>`	Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
`<Xn|SP>`	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
`<Wm>`	When `option<0>` is set to `0`, is the 32-bit name of the general-purpose index register, encoded in the "Rm" field.
`<Xm>`	When `option<0>` is set to `1`, is the 64-bit name of the general-purpose index register, encoded in the "Rm" field.
`<extend>`	Is the index extend/shift specifier, defaulting to LSL, and which must be omitted for the LSL option when <amount> is omitted. encoded in 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;
integer datasize = 8 << scale;
```

Operation

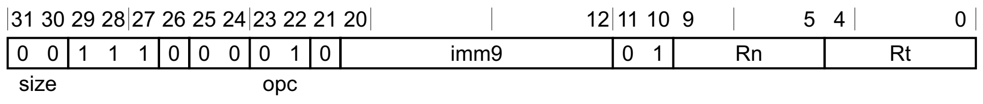
```
bits(64) offset = ExtendReg(m, extend_type, shift);
bits(64) address;
bits(datasize) data;

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

address = address + offset;

data = Mem[address, datasize DIV 8, AccType_NORMAL];
X[t] = ZeroExtend(data, regsize);
```

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.104 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* on page C1-143.

Post-index



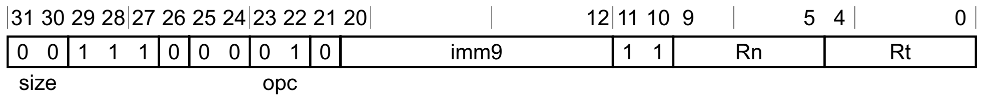
Post-index variant

```
LDRB <Wt>, [<Xn|SP>], #<simm>
```

Decode for this encoding

```
boolean wback = TRUE;
boolean postindex = TRUE;
bits(64) offset = SignExtend(imm9, 64);
```

Pre-index



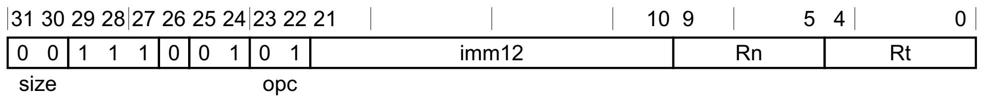
Pre-index variant

```
LDRB <Wt>, [<Xn|SP>, #<simm>]!
```

Decode for this encoding

```
boolean wback = TRUE;
boolean postindex = FALSE;
bits(64) offset = SignExtend(imm9, 64);
```

Unsigned offset



Unsigned offset variant

```
LDRB <Wt>, [<Xn|SP>{, #<pimm>}]
```

Decode for this encoding

```
boolean wback = FALSE;
boolean postindex = FALSE;
bits(64) offset = LSL(ZeroExtend(imm12, 64), 0);
```

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

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)* on page K1-6127.

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);
```

Operation for all encodings

```
bits(64) address;
bits(8) data;
boolean wb_unknown = FALSE;

if wback && n == t && n != 31 then
    c = ConstrainUnpredictable();
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;    // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;    // writeback is UNKNOWN
        when Constraint_UNDEF      UnallocatedEncoding();
        when Constraint_NOP        EndOfInstruction();

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

if !postindex then
    address = address + offset;

data = Mem[address, 1, AccType_NORMAL];
X[t] = ZeroExtend(data, 32);

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```

C6.2.105 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* on page C1-143.

31 30	29 28 27	26	25 24	23 22	21	20 16	15 13	12	11 10	9 5	4 0
0 0	1 1 1	0	0 0	0 1	1	Rm	option	S	1 0	Rn	Rt
size				opc							

Extended register 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 UnallocatedEncoding();    // 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);
bits(64) address;
bits(8) data;
```

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

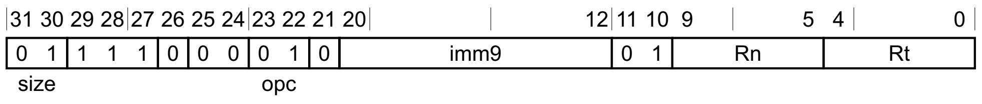
```
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

address = address + offset;

data = Mem[address, 1, AccType_NORMAL];
X[t] = ZeroExtend(data, 32);
```

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.106 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* on page C1-143.

Post-index



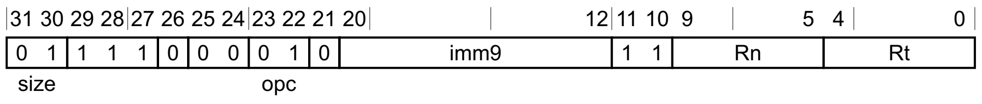
Post-index variant

```
LDRH <Wt>, [<Xn|SP>], #<simm>
```

Decode for this encoding

```
boolean wback = TRUE;
boolean postindex = TRUE;
bits(64) offset = SignExtend(imm9, 64);
```

Pre-index



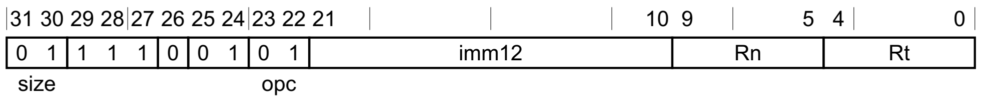
Pre-index variant

```
LDRH <Wt>, [<Xn|SP>, #<simm>]!
```

Decode for this encoding

```
boolean wback = TRUE;
boolean postindex = FALSE;
bits(64) offset = SignExtend(imm9, 64);
```

Unsigned offset



Unsigned offset variant

```
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)* on page K1-6127.

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

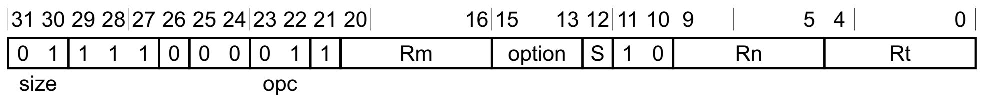
```
integer n = UInt(Rn);
integer t = UInt(Rt);
```

Operation for all encodings

```
bits(64) address;
bits(16) data;
boolean wb_unknown = FALSE;

if wback && n == t && n != 31 then
    c = ConstrainUnpredictable();
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;    // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;    // writeback is UNKNOWN
        when Constraint_UNDEF      UnallocatedEncoding();
        when Constraint_NOP        EndOfInstruction();

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

if !postindex then
    address = address + offset;

data = Mem[address, 2, AccType_NORMAL];
X[t] = ZeroExtend(data, 32);

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```

C6.2.107 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* on page C1-143.



32-bit variant

```
LDRH <Wt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

Decode for this encoding

```
if option<1> == '0' then UnallocatedEncoding();    // sub-word index
ExtendType extend_type = DecodeRegExtend(option);
integer shift = if S == '1' then 1 else 0;
```

Notes for all encodings

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Appendix K1 *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler symbols

`<Wt>`	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
`<Xn|SP>`	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
`<Wm>`	When `option<0>` is set to `0`, is the 32-bit name of the general-purpose index register, encoded in the "Rm" field.
`<Xm>`	When `option<0>` is set to `1`, is the 64-bit name of the general-purpose index register, encoded in the "Rm" field.
`<extend>`	Is the index extend/shift specifier, defaulting to LSL, and which must be omitted for the LSL option when <amount> is omitted. encoded in 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);
```

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
bits(64) offset = ExtendReg(m, extend_type, shift);
bits(64) address;
bits(16) data;

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

address = address + offset;

data = Mem[address, 2, AccType_NORMAL];
X[t] = ZeroExtend(data, 32);
```

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.108 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* on page C1-143.

Post-index

31	30	29	28	27	26	25	24	23	22	21	20 12	11	10	9 5	4 0
0	0	1	1	1	0	0	0	1	x	0	imm9	0	1	Rn	Rt
size								opc							

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

31	30	29	28	27	26	25	24	23	22	21	20 12	11	10	9 5	4 0
0	0	1	1	1	0	0	0	1	x	0	imm9	1	1	Rn	Rt
size								opc							

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

31	30	29	28	27	26	25	24	23	22	21 10	9 5	4 0
0	0	1	1	1	0	0	1	1	x	imm12	Rn	Rt
size								opc				

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

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)* on page K1-6127.

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;
```

Operation for all encodings

```
bits(64) address;
bits(8) data;
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == MemOp_LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable();
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;    // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;    // writeback is UNKNOWN
        when Constraint_UNDEF      UnallocatedEncoding();
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable();
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE    rt_unknown = FALSE;    // value stored is original value
        when Constraint_UNKNOWN rt_unknown = TRUE;    // value stored is UNKNOWN
        when Constraint_UNDEF   UnallocatedEncoding();
        when Constraint_NOP     EndOfInstruction();

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if !postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(8) UNKNOWN;
        else
            data = X[t];
        Mem[address, 1, AccType_NORMAL] = data;

    when MemOp_LOAD
        data = Mem[address, 1, AccType_NORMAL];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

    when MemOp_PREFETCH
        Prefetch(address, t<4:0>);

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.109 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* on page C1-143.

31 30	29 28 27	26	25 24	23 22	21	20 16	15 13	12	11 10	9 5	4 0
0 0	1 1 1	0	0 0	1 x	1	Rm	option	S	1 0	Rn	Rt
size				opc							

32-bit with extended register offset 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 UnallocatedEncoding();    // 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.

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

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;
```

Operation

```
bits(64) offset = ExtendReg(m, extend_type, 0);
bits(64) address;
bits(8) data;

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];

address = address + offset;

case memop of
    when MemOp_STORE
        data = X[t];
        Mem[address, 1, AccType_NORMAL] = data;

    when MemOp_LOAD
        data = Mem[address, 1, AccType_NORMAL];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

    when MemOp_PREFETCH
        Prefetch(address, t<4:0>);
```

C6.2.110 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* on page C1-143.

Post-index

31 30	29 28 27	26	25 24	23 22	21	20 … 12	11 10	9 … 5	4 … 0
0 1	1 1 1	0	0 0	1 x	0	imm9	0 1	Rn	Rt
size				opc					

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

31 30	29 28 27	26	25 24	23 22	21	20 … 12	11 10	9 … 5	4 … 0
0 1	1 1 1	0	0 0	1 x	0	imm9	1 1	Rn	Rt
size				opc					

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

31 30	29 28 27	26	25 24	23 22	21 … 10	9 … 5	4 … 0
0 1	1 1 1	0	0 1	1 x	imm12	Rn	Rt
size				opc			

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

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)* on page K1-6128.

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;
```

Operation for all encodings

```
bits(64) address;
bits(16) data;
boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == MemOp_LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable();
```

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

```
        assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_WBSUPPRESS wback = FALSE;    // writeback is suppressed
            when Constraint_UNKNOWN    wb_unknown = TRUE;    // writeback is UNKNOWN
            when Constraint_UNDEF      UnallocatedEncoding();
            when Constraint_NOP        EndOfInstruction();

    if memop == MemOp_STORE && wback && n == t && n != 31 then
        c = ConstrainUnpredictable();
        assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_NONE    rt_unknown = FALSE;    // value stored is original value
            when Constraint_UNKNOWN rt_unknown = TRUE;    // value stored is UNKNOWN
            when Constraint_UNDEF   UnallocatedEncoding();
            when Constraint_NOP     EndOfInstruction();

    if n == 31 then
        if memop != MemOp_PREFETCH then CheckSPAlignment();
        address = SP[];
    else
        address = X[n];

    if !postindex then
        address = address + offset;

    case memop of
        when MemOp_STORE
            if rt_unknown then
                data = bits(16) UNKNOWN;
            else
                data = X[t];
            Mem[address, 2, AccType_NORMAL] = data;

        when MemOp_LOAD
            data = Mem[address, 2, AccType_NORMAL];
            if signed then
                X[t] = SignExtend(data, regsize);
            else
                X[t] = ZeroExtend(data, regsize);

        when MemOp_PREFETCH
            Prefetch(address, t<4:0>);

    if wback then
        if wb_unknown then
            address = bits(64) UNKNOWN;
        elsif postindex then
            address = address + offset;
        if n == 31 then
            SP[] = address;
        else
            X[n] = address;
```

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.111 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* on page C1-143.

31 30	29 28 27	26	25 24	23 22	21	20 16	15 13	12	11 10	9 5	4 0
0 1	1 1 1	0	0 0	1 x	1	Rm	option	S	1 0	Rn	Rt
size				opc							

32-bit 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 UnallocatedEncoding();    // 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`

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

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;
```

Operation

```
bits(64) offset = ExtendReg(m, extend_type, shift);
bits(64) address;
bits(16) data;

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];

address = address + offset;

case memop of
    when MemOp_STORE
        data = X[t];
        Mem[address, 2, AccType_NORMAL] = data;

    when MemOp_LOAD
        data = Mem[address, 2, AccType_NORMAL];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

    when MemOp_PREFETCH
        Prefetch(address, t<4:0>);
```

C6.2.112 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* on page C1-143.

Post-index

31 30	29 28 27	26	25 24	23 22	21	20 ... 12	11 10	9 ... 5	4 ... 0
1 0	1 1 1	0	0 0	1 0	0	imm9	0 1	Rn	Rt
size				opc					

Post-index variant

```
LDRSW <Xt>, [<Xn|SP>], #<simm>
```

Decode for this encoding

```
boolean wback = TRUE;
boolean postindex = TRUE;
bits(64) offset = SignExtend(imm9, 64);
```

Pre-index

31 30	29 28 27	26	25 24	23 22	21	20 ... 12	11 10	9 ... 5	4 ... 0
1 0	1 1 1	0	0 0	1 0	0	imm9	1 1	Rn	Rt
size				opc					

Pre-index variant

```
LDRSW <Xt>, [<Xn|SP>, #<simm>]!
```

Decode for this encoding

```
boolean wback = TRUE;
boolean postindex = FALSE;
bits(64) offset = SignExtend(imm9, 64);
```

Unsigned offset

31 30	29 28 27	26	25 24	23 22	21 ... 10	9 ... 5	4 ... 0
1 0	1 1 1	0	0 1	1 0	imm12	Rn	Rt
size				opc			

Unsigned offset variant

```
LDRSW <Xt>, [<Xn|SP>{, #<pimm>}]
```

Decode for this encoding

```
boolean wback = FALSE;
boolean postindex = FALSE;
bits(64) offset = LSL(ZeroExtend(imm12, 64), 2);
```

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

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)* on page K1-6128.

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);
```

Operation for all encodings

```
bits(64) address;
bits(32) data;
boolean wb_unknown = FALSE;

if wback && n == t && n != 31 then
    c = ConstrainUnpredictable();
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;    // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;    // writeback is UNKNOWN
        when Constraint_UNDEF      UnallocatedEncoding();
        when Constraint_NOP        EndOfInstruction();

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

if !postindex then
    address = address + offset;

data = Mem[address, 4, AccType_NORMAL];
X[t] = SignExtend(data, 64);
if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```

C6.2.113 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* on page C1-143.

31	30	29	28	27	26	25	24	23 … 5	4 … 0
1	0	0	1	1	0	0	0	imm19	Rt
opc									

Literal variant

```
LDRSW <Xt>, <label>
```

Decode for this encoding

```
integer t = UInt(Rt);
bits(64) offset;

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

Assembler symbols

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

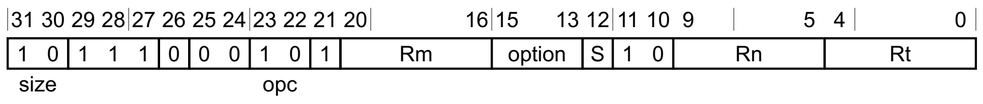
Operation

```
bits(64) address = PC[] + offset;
bits(32) data;

data = Mem[address, 4, AccType_NORMAL];
X[t] = SignExtend(data, 64);
```

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.114 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* on page C1-143.



64-bit variant

```
LDRSW <Xt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

Decode for this encoding

```
if option<1> == '0' then UnallocatedEncoding();    // 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);
bits(64) address;
bits(32) data;
```

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

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

address = address + offset;

data = Mem[address, 4, AccType_NORMAL];
X[t] = SignExtend(data, 64);
```

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.115 LDSETB, LDSETAB, LDSETALB, LDSETLB

Atomic bit set on byte in memory atomically loads an 8-bit byte from memory, performs a bitwise OR with the value held in a register on it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not `WZR`, `LDSETAB` and `LDSETALB` load from memory with acquire semantics.
- `LDSETLB` and `LDSETALB` store to memory with release semantics.
- `LDSETB` has no memory ordering requirements.

For more information about memory ordering semantics see *Load-Acquire, Store-Release* on page B2-94.

For information about memory accesses see *Load/Store addressing modes* on page C1-143.

ARMv8.1

31 30	29 28 27	26	25 24	23	22	21	20 16	15	14 12	11 10	9 5	4 0
0 0	1 1 1	0	0 0	A	R	1	Rs	0	0 1 1	0 0	Rn	Rt
size		V						o3	opc			

Acquire variant

Applies when `A == 1 && R == 0`.

```
LDSETAB <Ws>, <Wt>, [<Xn|SP>]
```

Acquire and release variant

Applies when `A == 1 && R == 1`.

```
LDSETALB <Ws>, <Wt>, [<Xn|SP>]
```

No memory ordering variant

Applies when `A == 0 && R == 0 && Rt != 11111`.

```
LDSETB <Ws>, <Wt>, [<Xn|SP>]
```

Release variant

Applies when `A == 0 && R == 1 && Rt != 11111`.

```
LDSETLB <Ws>, <Wt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !HaveAtomicExt() then UnallocatedEncoding();
integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
```

Assembler symbols

`<Ws>`	Is the 32-bit name of the general-purpose register 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;
bits(8) result;

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, 1, ldacctype];

result = data OR value;
// All observers in the shareability domain observe the
// following load and store atomically.
Mem[address, 1, stacctype] = result;

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

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.116 LDSETH, LDSETAH, LDSETALH, LDSETLH

Atomic bit set on halfword in memory atomically loads a 16-bit halfword from memory, performs a bitwise OR with the value held in a register on it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not `WZR`, `LDSETAH` and `LDSETALH` load from memory with acquire semantics.
- `LDSETLH` and `LDSETALH` store to memory with release semantics.
- `LDSETH` has no memory ordering requirements.

For more information about memory ordering semantics see *Load-Acquire, Store-Release* on page B2-94.

For information about memory accesses see *Load/Store addressing modes* on page C1-143.

ARMv8.1

31 30	29 28 27	26	25 24	23	22	21	20 16	15	14 12	11 10	9 5	4 0
0 1	1 1 1	0	0 0	A	R	1	Rs	0	0 1 1	0 0	Rn	Rt
size		V						o3	opc			

Acquire variant

Applies when `A == 1 && R == 0`.

```
LDSETAH <Ws>, <Wt>, [<Xn|SP>]
```

Acquire and release variant

Applies when `A == 1 && R == 1`.

```
LDSETALH <Ws>, <Wt>, [<Xn|SP>]
```

No memory ordering variant

Applies when `A == 0 && R == 0 && Rt != 11111`.

```
LDSETH <Ws>, <Wt>, [<Xn|SP>]
```

Release variant

Applies when `A == 0 && R == 1 && Rt != 11111`.

```
LDSETLH <Ws>, <Wt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !HaveAtomicExt() then UnallocatedEncoding();
integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
```

Assembler symbols

`<Ws>`	Is the 32-bit name of the general-purpose register 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.

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

`<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;
bits(16) result;

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, 2, ldacctype];

result = data OR value;
// All observers in the shareability domain observe the
// following load and store atomically.
Mem[address, 2, stacctype] = result;

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

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.117 LDSET, LDSETA, LDSETAL, LDSETL

Atomic bit set on word or doubleword in memory atomically loads a 32-bit word or 64-bit doubleword from memory, performs a bitwise OR with the value held in a register on it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not one of `WZR` or `XZR`, `LDSETA` and `LDSETAL` load from memory with acquire semantics.
- `LDSETL` and `LDSETAL` store to memory with release semantics.
- `LDSET` has no memory ordering requirements.

For more information about memory ordering semantics see *Load-Acquire, Store-Release* on page B2-94.

For information about memory accesses see *Load/Store addressing modes* on page C1-143.

ARMv8.1

31	30	29	28	27	26	25	24	23	22	21	20 16	15	14	13	12	11	10	9 5	4 0
1	x	1	1	1	0	0	0	A	R	1	Rs	0	0	1	1	0	0	Rn	Rt
size					V							o3	opc						

32-bit, acquire variant

Applies when `size == 10 && A == 1 && R == 0`.

```
LDSETA <Ws>, <Wt>, [<Xn|SP>]
```

32-bit, acquire and release variant

Applies when `size == 10 && A == 1 && R == 1`.

```
LDSETAL <Ws>, <Wt>, [<Xn|SP>]
```

32-bit, no memory ordering variant

Applies when `size == 10 && A == 0 && R == 0 && Rt != 11111`.

```
LDSET <Ws>, <Wt>, [<Xn|SP>]
```

32-bit, release variant

Applies when `size == 10 && A == 0 && R == 1 && Rt != 11111`.

```
LDSETL <Ws>, <Wt>, [<Xn|SP>]
```

64-bit, acquire variant

Applies when `size == 11 && A == 1 && R == 0`.

```
LDSETA <Xs>, <Xt>, [<Xn|SP>]
```

64-bit, acquire and release variant

Applies when `size == 11 && A == 1 && R == 1`.

```
LDSETAL <Xs>, <Xt>, [<Xn|SP>]
```

64-bit, no memory ordering variant

Applies when `size == 11 && A == 0 && R == 0 && Rt != 11111`.

```
LDSET <Xs>, <Xt>, [<Xn|SP>]
```

64-bit, release variant

Applies when `size == 11 && A == 0 && R == 1 && Rt != 11111`.

```
LDSETL <Xs>, <Xt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !HaveAtomicExt() then UnallocatedEncoding();
integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
```

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;
bits(datasize) result;

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, datasize DIV 8, ldacctype];

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

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

C6.2.118 LDSMAXB, LDSMAXAB, LDSMAXALB, LDSMAXLB

Atomic signed maximum on byte in memory atomically loads an 8-bit byte from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as signed numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not `WZR`, `LDSMAXAB` and `LDSMAXALB` load from memory with acquire semantics.
- `LDSMAXLB` and `LDSMAXALB` store to memory with release semantics.
- `LDSMAXB` has no memory ordering requirements.

For more information about memory ordering semantics see *Load-Acquire, Store-Release* on page B2-94.

For information about memory accesses see *Load/Store addressing modes* on page C1-143.

ARMv8.1

31 30	29 28 27	26	25 24	23	22	21	20 16	15	14 12	11 10	9 5	4 0
0 0	1 1 1	0	0 0	A	R	1	Rs	0	1 0 0	0 0	Rn	Rt
size		V						o3	opc			

Acquire variant

Applies when `A == 1 && R == 0`.

```
LDSMAXAB <Ws>, <Wt>, [<Xn|SP>]
```

Acquire and release variant

Applies when `A == 1 && R == 1`.

```
LDSMAXALB <Ws>, <Wt>, [<Xn|SP>]
```

No memory ordering variant

Applies when `A == 0 && R == 0 && Rt != 11111`.

```
LDSMAXB <Ws>, <Wt>, [<Xn|SP>]
```

Release variant

Applies when `A == 0 && R == 1 && Rt != 11111`.

```
LDSMAXLB <Ws>, <Wt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !HaveAtomicExt() then UnallocatedEncoding();
integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
```

Assembler symbols

`<Ws>`	Is the 32-bit name of the general-purpose register 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.

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

`<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;
bits(8) result;

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, 1, ldacctype];

result = if SInt(data) > SInt(value) then data else value;
// All observers in the shareability domain observe the
// following load and store atomically.
Mem[address, 1, stacctype] = result;

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

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.119 LDSMAXH, LDSMAXAH, LDSMAXALH, LDSMAXLH

Atomic signed maximum on halfword in memory atomically loads a 16-bit halfword from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as signed numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not `WZR`, `LDSMAXAH` and `LDSMAXALH` load from memory with acquire semantics.
- `LDSMAXLH` and `LDSMAXALH` store to memory with release semantics.
- `LDSMAXH` has no memory ordering requirements.

For more information about memory ordering semantics see *Load-Acquire, Store-Release* on page B2-94.

For information about memory accesses see *Load/Store addressing modes* on page C1-143.

ARMv8.1

31 30	29 28 27	26	25 24	23	22	21	20 16	15	14 12	11 10	9 5	4 0
0 1	1 1 1	0	0 0	A	R	1	Rs	0	1 0 0	0 0	Rn	Rt
size		V						o3	opc			

Acquire variant

Applies when `A == 1 && R == 0`.

`LDSMAXAH <Ws>, <Wt>, [<Xn|SP>]`

Acquire and release variant

Applies when `A == 1 && R == 1`.

`LDSMAXALH <Ws>, <Wt>, [<Xn|SP>]`

No memory ordering variant

Applies when `A == 0 && R == 0 && Rt != 11111`.

`LDSMAXH <Ws>, <Wt>, [<Xn|SP>]`

Release variant

Applies when `A == 0 && R == 1 && Rt != 11111`.

`LDSMAXLH <Ws>, <Wt>, [<Xn|SP>]`

Decode for all variants of this encoding

```
if !HaveAtomicExt() then UnallocatedEncoding();
integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
```

Assembler symbols

`<Ws>`	Is the 32-bit name of the general-purpose register 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;
bits(16) result;

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, 2, ldacctype];

result = if SInt(data) > SInt(value) then data else value;
// All observers in the shareability domain observe the
// following load and store atomically.
Mem[address, 2, stacctype] = result;

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

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.120 LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL

Atomic signed maximum on word or doubleword in memory atomically loads a 32-bit word or 64-bit doubleword from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as signed numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not one of `WZR` or `XZR`, `LDSMAXA` and `LDSMAXAL` load from memory with acquire semantics.
- `LDSMAXL` and `LDSMAXAL` store to memory with release semantics.
- `LDSMAX` has no memory ordering requirements.

For more information about memory ordering semantics see *Load-Acquire, Store-Release* on page B2-94.

For information about memory accesses see *Load/Store addressing modes* on page C1-143.

ARMv8.1

31 30	29 28 27	26	25 24	23	22	21	20 … 16	15	14 … 12	11 10	9 … 5	4 … 0
1 x	1 1 1	0	0 0	A	R	1	Rs	0	1 0 0	0 0	Rn	Rt
size		V						o3	opc			

32-bit, acquire variant

Applies when `size == 10 && A == 1 && R == 0`.

```
LDSMAXA <Ws>, <Wt>, [<Xn|SP>]
```

32-bit, acquire and release variant

Applies when `size == 10 && A == 1 && R == 1`.

```
LDSMAXAL <Ws>, <Wt>, [<Xn|SP>]
```

32-bit, no memory ordering variant

Applies when `size == 10 && A == 0 && R == 0 && Rt != 11111`.

```
LDSMAX <Ws>, <Wt>, [<Xn|SP>]
```

32-bit, release variant

Applies when `size == 10 && A == 0 && R == 1 && Rt != 11111`.

```
LDSMAXL <Ws>, <Wt>, [<Xn|SP>]
```

64-bit, acquire variant

Applies when `size == 11 && A == 1 && R == 0`.

```
LDSMAXA <Xs>, <Xt>, [<Xn|SP>]
```

64-bit, acquire and release variant

Applies when `size == 11 && A == 1 && R == 1`.

```
LDSMAXAL <Xs>, <Xt>, [<Xn|SP>]
```

64-bit, no memory ordering variant

Applies when `size == 11 && A == 0 && R == 0 && Rt != 11111`.

```
LDSMAX <Xs>, <Xt>, [<Xn|SP>]
```

64-bit, release variant

Applies when `size == 11 && A == 0 && R == 1 && Rt != 11111`.

```
LDSMAXL <Xs>, <Xt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !HaveAtomicExt() then UnallocatedEncoding();
integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
```

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;
bits(datasize) result;

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, datasize DIV 8, ldacctype];

result = if SInt(data) > SInt(value) then data else value;
// All observers in the shareability domain observe the
// following load and store atomically.
Mem[address, datasize DIV 8, stacctype] = result;

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

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.121 LDSMINB, LDSMINAB, LDSMINALB, LDSMINLB

Atomic signed minimum on byte in memory atomically loads an 8-bit byte from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as signed numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not `WZR`, `LDSMINAB` and `LDSMINALB` load from memory with acquire semantics.
- `LDSMINLB` and `LDSMINALB` store to memory with release semantics.
- `LDSMINB` has no memory ordering requirements.

For more information about memory ordering semantics see *Load-Acquire, Store-Release* on page B2-94.

For information about memory accesses see *Load/Store addressing modes* on page C1-143.

ARMv8.1

31 30	29 28 27	26	25 24	23	22	21	20 … 16	15	14 … 12	11 10	9 … 5	4 … 0
0 0	1 1 1	0	0 0	A	R	1	Rs	0	1 0 1	0 0	Rn	Rt
size		V						o3	opc			

Acquire variant

Applies when `A == 1 && R == 0`.

```
LDSMINAB <Ws>, <Wt>, [<Xn|SP>]
```

Acquire and release variant

Applies when `A == 1 && R == 1`.

```
LDSMINALB <Ws>, <Wt>, [<Xn|SP>]
```

No memory ordering variant

Applies when `A == 0 && R == 0 && Rt != 11111`.

```
LDSMINB <Ws>, <Wt>, [<Xn|SP>]
```

Release variant

Applies when `A == 0 && R == 1 && Rt != 11111`.

```
LDSMINLB <Ws>, <Wt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !HaveAtomicExt() then UnallocatedEncoding();
integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
```

Assembler symbols

`<Ws>`	Is the 32-bit name of the general-purpose register 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.

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

`<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;
bits(8) result;

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, 1, ldacctype];

result = if SInt(data) > SInt(value) then value else data;
// All observers in the shareability domain observe the
// following load and store atomically.
Mem[address, 1, stacctype] = result;

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

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.122 LDSMINH, LDSMINAH, LDSMINALH, LDSMINLH

Atomic signed minimum on halfword in memory atomically loads a 16-bit halfword from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as signed numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not `WZR`, `LDSMINAH` and `LDSMINALH` load from memory with acquire semantics.
- `LDSMINLH` and `LDSMINALH` store to memory with release semantics.
- `LDSMINH` has no memory ordering requirements.

For more information about memory ordering semantics see *Load-Acquire, Store-Release* on page B2-94.

For information about memory accesses see *Load/Store addressing modes* on page C1-143.

ARMv8.1

31 30	29 28 27	26	25 24	23	22	21	20 16	15	14 12	11 10	9 5	4 0
0 1	1 1 1	0	0 0	A	R	1	Rs	0	1 0 1	0 0	Rn	Rt
size		V						o3	opc			

Acquire variant

Applies when `A == 1 && R == 0`.

```
LDSMINAH <Ws>, <Wt>, [<Xn|SP>]
```

Acquire and release variant

Applies when `A == 1 && R == 1`.

```
LDSMINALH <Ws>, <Wt>, [<Xn|SP>]
```

No memory ordering variant

Applies when `A == 0 && R == 0 && Rt != 11111`.

```
LDSMINH <Ws>, <Wt>, [<Xn|SP>]
```

Release variant

Applies when `A == 0 && R == 1 && Rt != 11111`.

```
LDSMINLH <Ws>, <Wt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !HaveAtomicExt() then UnallocatedEncoding();
integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
```

Assembler symbols

`<Ws>`	Is the 32-bit name of the general-purpose register 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;
bits(16) result;

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, 2, ldacctype];

result = if SInt(data) > SInt(value) then value else data;
// All observers in the shareability domain observe the
// following load and store atomically.
Mem[address, 2, stacctype] = result;

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

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.123 LDSMIN, LDSMINA, LDSMINAL, LDSMINL

Atomic signed minimum on word or doubleword in memory atomically loads a 32-bit word or 64-bit doubleword from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as signed numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not one of `WZR` or `XZR`, `LDSMINA` and `LDSMINAL` load from memory with acquire semantics.
- `LDSMINL` and `LDSMINAL` store to memory with release semantics.
- `LDSMIN` has no memory ordering requirements.

For more information about memory ordering semantics see *Load-Acquire, Store-Release* on page B2-94.

For information about memory accesses see *Load/Store addressing modes* on page C1-143.

ARMv8.1

31 30	29 28 27	26	25 24	23	22	21	20 16	15	14 12	11 10	9 5	4 0
1 x	1 1 1	0	0 0	A	R	1	Rs	0	1 0 1	0 0	Rn	Rt
size		V						o3	opc			

32-bit, acquire variant

Applies when `size == 10 && A == 1 && R == 0`.

```
LDSMINA <Ws>, <Wt>, [<Xn|SP>]
```

32-bit, acquire and release variant

Applies when `size == 10 && A == 1 && R == 1`.

```
LDSMINAL <Ws>, <Wt>, [<Xn|SP>]
```

32-bit, no memory ordering variant

Applies when `size == 10 && A == 0 && R == 0 && Rt != 11111`.

```
LDSMIN <Ws>, <Wt>, [<Xn|SP>]
```

32-bit, release variant

Applies when `size == 10 && A == 0 && R == 1 && Rt != 11111`.

```
LDSMINL <Ws>, <Wt>, [<Xn|SP>]
```

64-bit, acquire variant

Applies when `size == 11 && A == 1 && R == 0`.

```
LDSMINA <Xs>, <Xt>, [<Xn|SP>]
```

64-bit, acquire and release variant

Applies when `size == 11 && A == 1 && R == 1`.

```
LDSMINAL <Xs>, <Xt>, [<Xn|SP>]
```

64-bit, no memory ordering variant

Applies when `size == 11 && A == 0 && R == 0 && Rt != 11111`.

```
LDSMIN <Xs>, <Xt>, [<Xn|SP>]
```

64-bit, release variant

Applies when `size == 11 && A == 0 && R == 1 && Rt != 11111`.

```
LDSMINL <Xs>, <Xt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !HaveAtomicExt() then UnallocatedEncoding();
integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
```

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;
bits(datasize) result;

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, datasize DIV 8, ldacctype];

result = if SInt(data) > SInt(value) then value else data;
// All observers in the shareability domain observe the
// following load and store atomically.
Mem[address, datasize DIV 8, stacctype] = result;

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

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.124 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.

The memory is restricted as if execution is at EL0 when:

- Executing at EL1.
- Executing at EL2, in ARMv8.1, with HCR_EL2.{E2H, TGE} set to {1, 1}.

Otherwise, the access permission is for the Exception level at which the instruction is executed. For information about memory accesses, see *Load/Store addressing modes* on page C1-143.

31 30	29 28 27	26	25 24	23 22	21	20 12	11 10	9 5	4 0
1 x	1 1 1	0	0 0	0 1	0	imm9	1 0	Rn	Rt
size				opc					

32-bit 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;
integer datasize = 8 << scale;
```

Operation

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

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

address = address + offset;

data = Mem[address, datasize DIV 8, AccType_UNPRIV];
X[t] = ZeroExtend(data, regsize);
```

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.125 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.

The memory is restricted as if execution is at EL0 when:

- Executing at EL1.
- Executing at EL2, in ARMv8.1, with HCR_EL2.{E2H, TGE} set to {1, 1}.

Otherwise, the access permission is for the Exception level at which the instruction is executed. For information about memory accesses, see *Load/Store addressing modes* on page C1-143.

31 30	29 28 27	26	25 24	23 22	21	20 … 12	11 10	9 … 5	4 … 0
0 0	1 1 1	0	0 0	0 1	0	imm9	1 0	Rn	Rt
size				opc					

Unscaled offset variant

```
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);
```

Operation

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

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

address = address + offset;

data = Mem[address, 1, AccType_UNPRIV];
X[t] = ZeroExtend(data, 32);
```

C6.2.126 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.

The memory is restricted as if execution is at EL0 when:

- Executing at EL1.
- Executing at EL2, in ARMv8.1, with HCR_EL2.{E2H, TGE} set to {1, 1}.

Otherwise, the access permission is for the Exception level at which the instruction is executed. For information about memory accesses, see *Load/Store addressing modes* on page C1-143.

31 30	29 28 27	26	25 24	23 22	21	20 … 12	11 10	9 … 5	4 … 0
0 1	1 1 1	0	0 0	0 1	0	imm9	1 0	Rn	Rt
size				opc					

Unscaled offset variant

```
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);
```

Operation

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

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

address = address + offset;

data = Mem[address, 2, AccType_UNPRIV];
X[t] = ZeroExtend(data, 32);
```

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.127 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.

The memory is restricted as if execution is at EL0 when:

- Executing at EL1.
- Executing at EL2, in ARMv8.1, with HCR_EL2.{E2H, TGE} set to {1, 1}.

Otherwise, the access permission is for the Exception level at which the instruction is executed. For information about memory accesses, see *Load/Store addressing modes* on page C1-143.

31 30	29 28 27	26	25 24	23 22	21	20 … 12	11 10	9 … 5	4 … 0
0 0	1 1 1	0	0 0	1 x	0	imm9	1 0	Rn	Rt
size				opc					

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;
```

Operation

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

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];

address = address + offset;

case memop of
    when MemOp_STORE
        data = X[t];
        Mem[address, 1, AccType_UNPRIV] = data;

    when MemOp_LOAD
        data = Mem[address, 1, AccType_UNPRIV];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

    when MemOp_PREFETCH
        Prefetch(address, t<4:0>);
```

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.128 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.

The memory is restricted as if execution is at EL0 when:

- Executing at EL1.
- Executing at EL2, in ARMv8.1, with HCR_EL2.{E2H, TGE} set to {1, 1}.

Otherwise, the access permission is for the Exception level at which the instruction is executed. For information about memory accesses, see *Load/Store addressing modes* on page C1-143.

31 30	29 28 27	26	25 24	23 22	21	20 … 12	11 10	9 … 5	4 … 0
0 1	1 1 1	0	0 0	1 x	0	imm9	1 0	Rn	Rt
size				opc					

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;
```

Operation

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

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];

address = address + offset;

case memop of
    when MemOp_STORE
        data = X[t];
        Mem[address, 2, AccType_UNPRIV] = data;

    when MemOp_LOAD
        data = Mem[address, 2, AccType_UNPRIV];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

    when MemOp_PREFETCH
        Prefetch(address, t<4:0>);
```

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.129 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.

The memory is restricted as if execution is at EL0 when:

- Executing at EL1.
- Executing at EL2, in ARMv8.1, with HCR_EL2.{E2H, TGE} set to {1, 1}.

Otherwise, the access permission is for the Exception level at which the instruction is executed. For information about memory accesses, see *Load/Store addressing modes* on page C1-143.

31 30	29 28 27	26	25 24	23 22	21	20 12	11 10	9 5	4 0
1 0	1 1 1	0	0 0	1 0	0	imm9	1 0	Rn	Rt
size				opc					

Unscaled offset variant

```
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);
```

Operation

```
bits(64) address;
bits(32) data;

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

address = address + offset;

data = Mem[address, 4, AccType_UNPRIV];
X[t] = SignExtend(data, 64);
```

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.130 LDUMAXB, LDUMAXAB, LDUMAXALB, LDUMAXLB

Atomic unsigned maximum on byte in memory atomically loads an 8-bit byte from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as unsigned numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not `WZR`, `LDUMAXAB` and `LDUMAXALB` load from memory with acquire semantics.
- `LDUMAXLB` and `LDUMAXALB` store to memory with release semantics.
- `LDUMAXB` has no memory ordering requirements.

For more information about memory ordering semantics see *Load-Acquire, Store-Release* on page B2-94.

For information about memory accesses see *Load/Store addressing modes* on page C1-143.

ARMv8.1

31 30	29 28 27	26	25 24	23	22	21	20 16	15	14 12	11 10	9 5	4 0
0 0	1 1 1	0	0 0	A	R	1	Rs	0	1 1 0	0 0	Rn	Rt
size		V						o3	opc			

Acquire variant

Applies when `A == 1 && R == 0`.

`LDUMAXAB <Ws>, <Wt>, [<Xn|SP>]`

Acquire and release variant

Applies when `A == 1 && R == 1`.

`LDUMAXALB <Ws>, <Wt>, [<Xn|SP>]`

No memory ordering variant

Applies when `A == 0 && R == 0 && Rt != 11111`.

`LDUMAXB <Ws>, <Wt>, [<Xn|SP>]`

Release variant

Applies when `A == 0 && R == 1 && Rt != 11111`.

`LDUMAXLB <Ws>, <Wt>, [<Xn|SP>]`

Decode for all variants of this encoding

```
if !HaveAtomicExt() then UnallocatedEncoding();
integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
```

Assembler symbols

`<Ws>`	Is the 32-bit name of the general-purpose register 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.

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

`<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;
bits(8) result;

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, 1, ldacctype];

result = if UInt(data) > UInt(value) then data else value;
// All observers in the shareability domain observe the
// following load and store atomically.
Mem[address, 1, stacctype] = result;

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

C6.2.131 LDUMAXH, LDUMAXAH, LDUMAXALH, LDUMAXLH

Atomic unsigned maximum on halfword in memory atomically loads a 16-bit halfword from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as unsigned numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not `WZR`, `LDUMAXAH` and `LDUMAXALH` load from memory with acquire semantics.
- `LDUMAXLH` and `LDUMAXALH` store to memory with release semantics.
- `LDUMAXH` has no memory ordering requirements.

For more information about memory ordering semantics see *Load-Acquire, Store-Release* on page B2-94.

For information about memory accesses see *Load/Store addressing modes* on page C1-143.

ARMv8.1

31	30	29	28	27	26	25	24	23	22	21	20 16	15	14 12	11	10	9 5	4 0
0	1	1	1	1	0	0	0	A	R	1	Rs	0	1 1 0	0	0	Rn	Rt
size					V							o3	opc				

Acquire variant

Applies when `A == 1 && R == 0`.

```
LDUMAXAH <Ws>, <Wt>, [<Xn|SP>]
```

Acquire and release variant

Applies when `A == 1 && R == 1`.

```
LDUMAXALH <Ws>, <Wt>, [<Xn|SP>]
```

No memory ordering variant

Applies when `A == 0 && R == 0 && Rt != 11111`.

```
LDUMAXH <Ws>, <Wt>, [<Xn|SP>]
```

Release variant

Applies when `A == 0 && R == 1 && Rt != 11111`.

```
LDUMAXLH <Ws>, <Wt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !HaveAtomicExt() then UnallocatedEncoding();
integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
```

Assembler symbols

`<Ws>`	Is the 32-bit name of the general-purpose register 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.

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

`<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;
bits(16) result;

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, 2, ldacctype];

result = if UInt(data) > UInt(value) then data else value;
// All observers in the shareability domain observe the
// following load and store atomically.
Mem[address, 2, stacctype] = result;

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

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.132 LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL

Atomic unsigned maximum on word or doubleword in memory atomically loads a 32-bit word or 64-bit doubleword from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as unsigned numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not one of `WZR` or `XZR`, `LDUMAXA` and `LDUMAXAL` load from memory with acquire semantics.
- `LDUMAXL` and `LDUMAXAL` store to memory with release semantics.
- `LDUMAX` has no memory ordering requirements.

For more information about memory ordering semantics see *Load-Acquire, Store-Release* on page B2-94.

For information about memory accesses see *Load/Store addressing modes* on page C1-143.

ARMv8.1

31 30	29 28 27	26	25 24	23	22	21	20 … 16	15	14 … 12	11 10	9 … 5	4 … 0
1 x	1 1 1	0	0 0	A	R	1	Rs	0	1 1 0	0 0	Rn	Rt
size		V						o3	opc			

32-bit, acquire variant

Applies when `size == 10 && A == 1 && R == 0`.

```
LDUMAXA <Ws>, <Wt>, [<Xn|SP>]
```

32-bit, acquire and release variant

Applies when `size == 10 && A == 1 && R == 1`.

```
LDUMAXAL <Ws>, <Wt>, [<Xn|SP>]
```

32-bit, no memory ordering variant

Applies when `size == 10 && A == 0 && R == 0 && Rt != 11111`.

```
LDUMAX <Ws>, <Wt>, [<Xn|SP>]
```

32-bit, release variant

Applies when `size == 10 && A == 0 && R == 1 && Rt != 11111`.

```
LDUMAXL <Ws>, <Wt>, [<Xn|SP>]
```

64-bit, acquire variant

Applies when `size == 11 && A == 1 && R == 0`.

```
LDUMAXA <Xs>, <Xt>, [<Xn|SP>]
```

64-bit, acquire and release variant

Applies when `size == 11 && A == 1 && R == 1`.

```
LDUMAXAL <Xs>, <Xt>, [<Xn|SP>]
```

64-bit, no memory ordering variant

Applies when `size == 11 && A == 0 && R == 0 && Rt != 11111`.

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

```
LDUMAX <Xs>, <Xt>, [<Xn|SP>]
```

64-bit, release variant

Applies when `size == 11 && A == 0 && R == 1 && Rt != 11111`.

```
LDUMAXL <Xs>, <Xt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !HaveAtomicExt() then UnallocatedEncoding();
integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
```

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;
bits(datasize) result;

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, datasize DIV 8, ldacctype];

result = if UInt(data) > UInt(value) then data else value;
// All observers in the shareability domain observe the
// following load and store atomically.
Mem[address, datasize DIV 8, stacctype] = result;

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

C6.2.133 LDUMINB, LDUMINAB, LDUMINALB, LDUMINLB

Atomic unsigned minimum on byte in memory atomically loads an 8-bit byte from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as unsigned numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not `WZR`, `LDUMINAB` and `LDUMINALB` load from memory with acquire semantics.
- `LDUMINLB` and `LDUMINALB` store to memory with release semantics.
- `LDUMINB` has no memory ordering requirements.

For more information about memory ordering semantics see *Load-Acquire, Store-Release* on page B2-94.

For information about memory accesses see *Load/Store addressing modes* on page C1-143.

ARMv8.1

31 30	29 28 27	26	25 24	23	22	21	20 16	15	14 12	11 10	9 5	4 0
0 0	1 1 1	0	0 0	A	R	1	Rs	0	1 1 1	0 0	Rn	Rt
size		V						o3	opc			

Acquire variant

Applies when `A == 1 && R == 0`.

`LDUMINAB <Ws>, <Wt>, [<Xn|SP>]`

Acquire and release variant

Applies when `A == 1 && R == 1`.

`LDUMINALB <Ws>, <Wt>, [<Xn|SP>]`

No memory ordering variant

Applies when `A == 0 && R == 0 && Rt != 11111`.

`LDUMINB <Ws>, <Wt>, [<Xn|SP>]`

Release variant

Applies when `A == 0 && R == 1 && Rt != 11111`.

`LDUMINLB <Ws>, <Wt>, [<Xn|SP>]`

Decode for all variants of this encoding

```
if !HaveAtomicExt() then UnallocatedEncoding();
integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
```

Assembler symbols

`<Ws>`	Is the 32-bit name of the general-purpose register 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.

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

`<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;
bits(8) result;

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, 1, ldacctype];

result = if UInt(data) > UInt(value) then value else data;
// All observers in the shareability domain observe the
// following load and store atomically.
Mem[address, 1, stacctype] = result;

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

C6.2.134 LDUMINH, LDUMINAH, LDUMINALH, LDUMINLH

Atomic unsigned minimum on halfword in memory atomically loads a 16-bit halfword from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as unsigned numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not `WZR`, `LDUMINAH` and `LDUMINALH` load from memory with acquire semantics.
- `LDUMINLH` and `LDUMINALH` store to memory with release semantics.
- `LDUMINH` has no memory ordering requirements.

For more information about memory ordering semantics see *Load-Acquire, Store-Release* on page B2-94.

For information about memory accesses see *Load/Store addressing modes* on page C1-143.

ARMv8.1

31 30	29 28 27	26	25 24	23	22	21	20 16	15	14 12	11 10	9 5	4 0
0 1	1 1 1	0	0 0	A	R	1	Rs	0	1 1 1	0 0	Rn	Rt
size		V						o3	opc			

Acquire variant

Applies when `A == 1 && R == 0`.

```
LDUMINAH <Ws>, <Wt>, [<Xn|SP>]
```

Acquire and release variant

Applies when `A == 1 && R == 1`.

```
LDUMINALH <Ws>, <Wt>, [<Xn|SP>]
```

No memory ordering variant

Applies when `A == 0 && R == 0 && Rt != 11111`.

```
LDUMINH <Ws>, <Wt>, [<Xn|SP>]
```

Release variant

Applies when `A == 0 && R == 1 && Rt != 11111`.

```
LDUMINLH <Ws>, <Wt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !HaveAtomicExt() then UnallocatedEncoding();
integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
```

Assembler symbols

`<Ws>`	Is the 32-bit name of the general-purpose register 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.

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

`<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;
bits(16) result;

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, 2, ldacctype];

result = if UInt(data) > UInt(value) then value else data;
// All observers in the shareability domain observe the
// following load and store atomically.
Mem[address, 2, stacctype] = result;

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

C6.2.135 LDUMIN, LDUMINA, LDUMINAL, LDUMINL

Atomic unsigned minimum on word or doubleword in memory atomically loads a 32-bit word or 64-bit doubleword from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as unsigned numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not one of `WZR` or `XZR`, `LDUMINA` and `LDUMINAL` load from memory with acquire semantics.
- `LDUMINL` and `LDUMINAL` store to memory with release semantics.
- `LDUMIN` has no memory ordering requirements.

For more information about memory ordering semantics see *Load-Acquire, Store-Release* on page B2-94.

For information about memory accesses see *Load/Store addressing modes* on page C1-143.

ARMv8.1

31	30	29	28	27	26	25	24	23	22	21	20 16	15	14	13	12	11	10	9 5	4 0
1	x	1	1	1	0	0	0	A	R	1	Rs	0	1	1	1	0	0	Rn	Rt
size					V							o3	opc						

32-bit, acquire variant

Applies when `size == 10 && A == 1 && R == 0`.

```
LDUMINA <Ws>, <Wt>, [<Xn|SP>]
```

32-bit, acquire and release variant

Applies when `size == 10 && A == 1 && R == 1`.

```
LDUMINAL <Ws>, <Wt>, [<Xn|SP>]
```

32-bit, no memory ordering variant

Applies when `size == 10 && A == 0 && R == 0 && Rt != 11111`.

```
LDUMIN <Ws>, <Wt>, [<Xn|SP>]
```

32-bit, release variant

Applies when `size == 10 && A == 0 && R == 1 && Rt != 11111`.

```
LDUMINL <Ws>, <Wt>, [<Xn|SP>]
```

64-bit, acquire variant

Applies when `size == 11 && A == 1 && R == 0`.

```
LDUMINA <Xs>, <Xt>, [<Xn|SP>]
```

64-bit, acquire and release variant

Applies when `size == 11 && A == 1 && R == 1`.

```
LDUMINAL <Xs>, <Xt>, [<Xn|SP>]
```

64-bit, no memory ordering variant

Applies when `size == 11 && A == 0 && R == 0 && Rt != 11111`.

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

```
LDUMIN <Xs>, <Xt>, [<Xn|SP>]
```

64-bit, release variant

Applies when `size == 11 && A == 0 && R == 1 && Rt != 11111`.

```
LDUMINL <Xs>, <Xt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !HaveAtomicExt() then UnallocatedEncoding();
integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
```

Assembler symbols

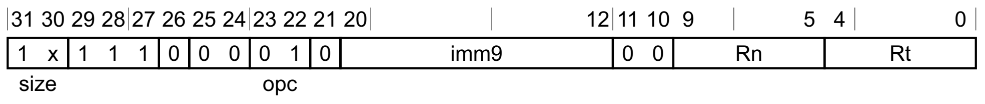
`<Ws>`	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
`<Wt>`	Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
`<Xs>`	Is the 64-bit name of the general-purpose 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;
bits(datasize) result;

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, datasize DIV 8, ldacctype];

result = if UInt(data) > UInt(value) then value else data;
// All observers in the shareability domain observe the
// following load and store atomically.
Mem[address, datasize DIV 8, stacctype] = result;

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

C6.2.136 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* on page C1-143.



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;
integer datasize = 8 << scale;
```

Operation

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

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

address = address + offset;
```

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

```
        data = Mem[address, datasize DIV 8, AccType_NORMAL];
        X[t] = ZeroExtend(data, regsize);
```

C6.2.137 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* on page C1-143.

31	30	29	28	27	26	25	24	23	22	21	20 … 12	11	10	9 … 5	4 … 0
0	0	1	1	1	0	0	0	0	1	0	imm9	0	0	Rn	Rt
size								opc							

Unscaled offset variant

```
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);
```

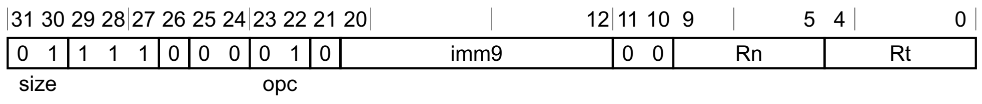
Operation

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

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

address = address + offset;

data = Mem[address, 1, AccType_NORMAL];
X[t] = ZeroExtend(data, 32);
```

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.138 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* on page C1-143.



Unscaled offset variant

```
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);
```

Operation

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

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

address = address + offset;

data = Mem[address, 2, AccType_NORMAL];
X[t] = ZeroExtend(data, 32);
```

C6.2.139 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* on page C1-143.

31 30	29 28 27	26	25 24	23 22	21	20 … 12	11 10	9 … 5	4 … 0
0 0	1 1 1	0	0 0	1 x	0	imm9	0 0	Rn	Rt
size				opc					

32-bit 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;
```

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

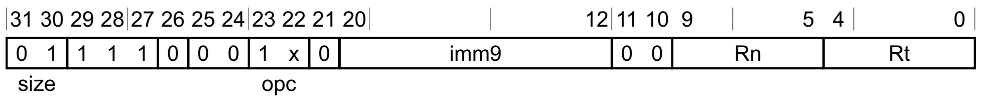
Operation

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

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];

address = address + offset;

case memop of
    when MemOp_STORE
        data = X[t];
        Mem[address, 1, AccType_NORMAL] = data;

    when MemOp_LOAD
        data = Mem[address, 1, AccType_NORMAL];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

    when MemOp_PREFETCH
        Prefetch(address, t<4:0>);
```

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.140 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* on page C1-143.



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;
```

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

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

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];

address = address + offset;

case memop of
    when MemOp_STORE
        data = X[t];
        Mem[address, 2, AccType_NORMAL] = data;

    when MemOp_LOAD
        data = Mem[address, 2, AccType_NORMAL];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

    when MemOp_PREFETCH
        Prefetch(address, t<4:0>);
```

C6.2.141 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* on page C1-143.

31	30	29	28	27	26	25	24	23	22	21	20 … 12	11	10	9 … 5	4 … 0
1	0	1	1	1	0	0	0	1	0	0	imm9	0	0	Rn	Rt
size								opc							

Unscaled offset variant

```
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);
```

Operation

```
bits(64) address;
bits(32) data;

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

address = address + offset;

data = Mem[address, 4, AccType_NORMAL];
X[t] = SignExtend(data, 64);
```

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.142 LDXP

Load Exclusive Pair of Registers derives an address from a base register value, loads two 32-bit words or two 64-bit doublewords from memory, and writes them to two registers. A 32-bit pair requires the address to be doubleword aligned and is single-copy atomic at doubleword granularity. A 64-bit pair requires the address to be quadword aligned and is single-copy atomic for each doubleword at doubleword granularity. The PE marks the physical address being accessed as an exclusive access. This exclusive access mark is checked by Store Exclusive instructions. See *Synchronization and semaphores* on page B2-121. For information about memory accesses see *Load/Store addressing modes* on page C1-143.

31	30	29 28 27 26 25 24	23	22	21	20 16	15	14 10	9 5	4 0
1	sz	0 0 1 0 0 0	0	1	1	(1) (1) (1) (1) (1)	0	Rt2	Rn	Rt
				L		Rs	o0			

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);

integer elsize = 32 << UInt(sz);
integer datasize = elsize * 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 *LDXP* on page K1-6128.

Assembler symbols

`<Wt1>`	Is the 32-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.
`<Wt2>`	Is the 32-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.
`<Xt1>`	Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.
`<Xt2>`	Is the 64-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.
`<Xn|SP>`	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;
```

```
boolean rt_unknown = FALSE;

if t == t2 then
    Constraint c = ConstrainUnpredictable();
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN rt_unknown = TRUE;    // result is UNKNOWN
        when Constraint_UNDEF   UnallocatedEncoding();
        when Constraint_NOP     EndOfInstruction();

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

// Tell the Exclusive Monitors to record a sequence of one or more atomic
// memory reads from virtual address range [address, address+dbytes-1].
// The Exclusive Monitor will only be set if all the reads are from the
// same dbytes-aligned physical address, to allow for the possibility of
// an atomicity break if the translation is changed between reads.
AArch64.SetExclusiveMonitors(address, dbytes);

if rt_unknown then
    // ConstrainedUNPREDICTABLE case
    X[t] = bits(datasize) UNKNOWN;
elsif elsize == 32 then
    // 32-bit load exclusive pair (atomic)
    data = Mem[address, dbytes, AccType_ATOMIC];
    if BigEndian() then
        X[t] = data<datasize-1:elsize>;
        X[t2] = data<elsize-1:0>;
    else
        X[t] = data<elsize-1:0>;
        X[t2] = data<datasize-1:elsize>;
else // elsize == 64
    // 64-bit load exclusive pair (not atomic),
    // but must be 128-bit aligned
    if address != Align(address, dbytes) then
        AArch64.Abort(address, AArch64.AlignmentFault(AccType_ATOMIC, FALSE, FALSE));
    X[t] = Mem[address, 8, AccType_ATOMIC];
    X[t2] = Mem[address+8, 8, AccType_ATOMIC];
```

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.143 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* on page B2-121. For information about memory accesses see *Load/Store addressing modes* on page C1-143.

31	30	29	28	27	26	25	24	23	22	21	20 … 16	15	14 … 10	9 … 5	4 … 0
1	x	0	0	1	0	0	0	0	1	0	(1) (1) (1) (1) (1)	0	(1) (1) (1) (1) (1)	Rn	Rt
size									L		Rs	o0	Rt2		

32-bit 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);

integer elsize = 8 << UInt(size);
integer regsize = if elsize == 64 then 64 else 32;
```

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;

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

// Tell the Exclusive Monitors to record a sequence of one or more atomic
// memory reads from virtual address range [address, address+dbytes-1].
// The Exclusive Monitor will only be set if all the reads are from the
// same dbytes-aligned physical address, to allow for the possibility of
// an atomicity break if the translation is changed between reads.
AArch64.SetExclusiveMonitors(address, dbytes);

data = Mem[address, dbytes, AccType_ATOMIC];
X[t] = ZeroExtend(data, regsize);
```

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.144 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* on page B2-121. For information about memory accesses see *Load/Store addressing modes* on page C1-143.

31 30	29 28 27 26 25 24	23	22	21	20 … 16	15	14 … 10	9 … 5	4 … 0
0 0	0 0 1 0 0 0	0	1	0	(1) (1) (1) (1) (1)	0	(1) (1) (1) (1) (1)	Rn	Rt
size			L		Rs	o0	Rt2		

No offset variant

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

Decode for this encoding

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

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;

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

// Tell the Exclusive Monitors to record a sequence of one or more atomic
// memory reads from virtual address range [address, address+dbytes-1].
// The Exclusive Monitor will only be set if all the reads are from the
// same dbytes-aligned physical address, to allow for the possibility of
// an atomicity break if the translation is changed between reads.
AArch64.SetExclusiveMonitors(address, 1);

data = Mem[address, 1, AccType_ATOMIC];
X[t] = ZeroExtend(data, 32);
```

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.145 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* on page B2-121. For information about memory accesses see *Load/Store addressing modes* on page C1-143.

31	30	29	28	27	26	25	24	23	22	21	20 16	15	14 10	9 5	4 0
0	1	0	0	1	0	0	0	0	1	0	(1) (1) (1) (1) (1)	0	(1) (1) (1) (1) (1)	Rn	Rt
size									L		Rs	o0	Rt2		

No offset variant

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

Decode for this encoding

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

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;

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

// Tell the Exclusive Monitors to record a sequence of one or more atomic
// memory reads from virtual address range [address, address+dbytes-1].
// The Exclusive Monitor will only be set if all the reads are from the
// same dbytes-aligned physical address, to allow for the possibility of
// an atomicity break if the translation is changed between reads.
AArch64.SetExclusiveMonitors(address, 2);

data = Mem[address, 2, AccType_ATOMIC];
X[t] = ZeroExtend(data, 32);
```

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.146 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 for this instruction.

31	30	29	28	27	26	25	24	23	22	21	20 16	15	14	13	12	11	10	9 5	4 0
sf	0	0	1	1	0	1	0	1	1	0	Rm	0	0	1	0	0	0	Rn	Rd
																op2			

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.

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.147 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 for this instruction.

31	30 29	28 27 26 25 24 23	22	21 16	15 10	9 5	4 0
sf	1 0	1 0 0 1 1 0	N	immr	!=x11111	Rn	Rd
	opc				imms		

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.

C6.2.148 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.

31	30	29	28 27 26 25 24 23 22 21	20 16	15 14 13 12	11 10	9 5	4 0
sf	0	0	1 1 0 1 0 1 1 0	Rm	0 0 1 0	0 0	Rn	Rd
						op2		

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);
integer datasize = if sf == '1' then 64 else 32;
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];

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

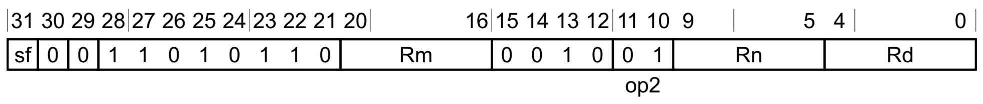
Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.149 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 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

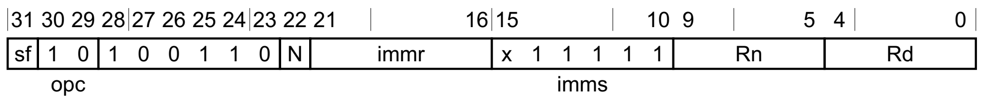
The description of LSRV gives the operational pseudocode for this instruction.

C6.2.150 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 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.

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.151 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.

31	30	29	28 27 26 25 24 23 22 21	20 … 16	15 14 13 12	11 10	9 … 5	4 … 0
sf	0	0	1 1 0 1 0 1 1 0	Rm	0 0 1 0	0 1	Rn	Rd
						op2		

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);
integer datasize = if sf == '1' then 64 else 32;
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

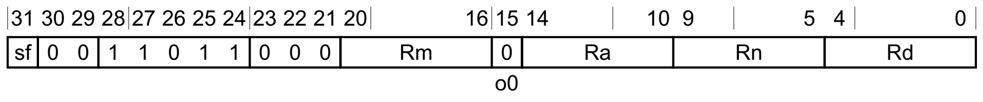
```
bits(datasize) result;
bits(datasize) operand2 = X[m];

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

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.152 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);
integer destsize = if sf == '1' then 64 else 32;
```

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.

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

<Xa>	Is the 64-bit name of the third general-purpose source register holding the addend, encoded in the "Ra" field.

Operation

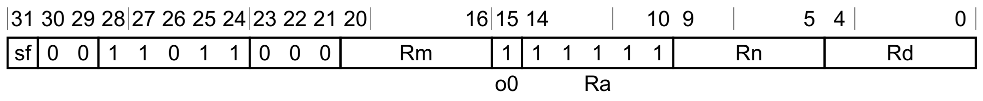
```
bits(destsize) operand1 = X[n];
bits(destsize) operand2 = X[m];
bits(destsize) operand3 = X[a];

integer result;

result = UInt(operand3) + (UInt(operand1) * UInt(operand2));

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

C6.2.153 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 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.

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.154 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 for this instruction.

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

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.

C6.2.155 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 for this instruction.

31	30 29	28 27 26 25 24 23	22 21	20 … 5	4 … 0
sf	0 0	1 0 0 1 0 1	hw	imm16	Rd
	opc				

32-bit variant

Applies when `sf == 0`.

```
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.

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.156 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 for this instruction.

31	30 29	28 27 26 25 24 23	22 21	20 … 5	4 … 0
sf	1 0	1 0 0 1 0 1	hw	imm16	Rd
	opc				

32-bit variant

Applies when `sf == 0`.

```
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

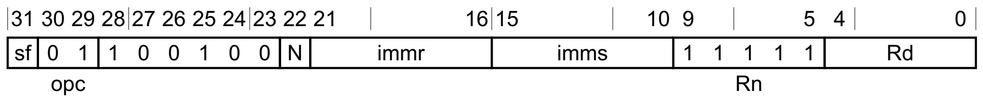
The description of MOVZ gives the operational pseudocode for this instruction.

C6.2.157 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 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.

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.158 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 for this instruction.

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

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.

C6.2.159 MOVK

Move wide with keep moves an optionally-shifted 16-bit immediate value into a register, keeping other bits unchanged.

31	30	29	28	27	26	25	24	23	22 21	20 5	4 0
sf	1	1	1	0	0	1	0	1	hw	imm16	Rd
	opc										

32-bit variant

Applies when `sf == 0`.

```
MOVK <Wd>, #<imm>{, LSL #<shift>}
```

64-bit variant

Applies when `sf == 1`.

```
MOVK <Xd>, #<imm>{, LSL #<shift>}
```

Decode for all variants of this encoding

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

if sf == '0' && hw<1> == '1' then UnallocatedEncoding();
pos = UInt(hw:'0000');
```

Assembler symbols

`<Wd>`	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
`<Xd>`	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
`<imm>`	Is the 16-bit unsigned immediate, in the range 0 to 65535, encoded in the "imm16" field.
`<shift>`	For the 32-bit variant: is the amount by which to shift the immediate left, either 0 (the default) or 16, encoded in the "hw" field as <shift>/16. For the 64-bit variant: is the amount by which to shift the immediate left, either 0 (the default), 16, 32 or 48, encoded in the "hw" field as <shift>/16.

Operation

```
bits(datasize) result;

result = X[d];
result<pos+15:pos> = imm16;
X[d] = result;
```

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.160 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.

31	30 29	28 27 26 25 24 23	22 21	20 … 5	4 … 0
sf	0 0	1 0 0 1 0 1	hw	imm16	Rd
	opc				

32-bit variant

Applies when `sf == 0`.

```
MOVN <Wd>, #<imm>{, LSL #<shift>}
```

64-bit variant

Applies when `sf == 1`.

```
MOVN <Xd>, #<imm>{, LSL #<shift>}
```

Decode for all variants of this encoding

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

if sf == '0' && hw<1> == '1' then UnallocatedEncoding();
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();
```

```
result<pos+15:pos> = imm16;
result = NOT(result);
X[d] = result;
```

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.161 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.

31	30	29	28	27	26	25	24	23	22 21	20 … 5	4 … 0
sf	1	0	1	0	0	1	0	1	hw	imm16	Rd
	opc	opc									

32-bit variant

Applies when `sf == 0`.

```
MOVZ <Wd>, #<imm>{, LSL #<shift>}
```

64-bit variant

Applies when `sf == 1`.

```
MOVZ <Xd>, #<imm>{, LSL #<shift>}
```

Decode for all variants of this encoding

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

if sf == '0' && hw<1> == '1' then UnallocatedEncoding();
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

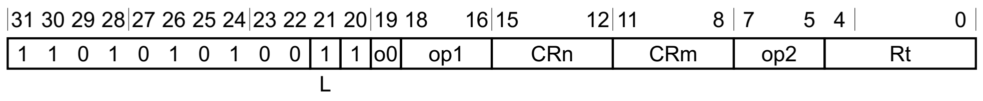
```
bits(datasize) result;

result = Zeros();
```

```
result<pos+15:pos> = imm16;
X[d] = result;
```

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.162 MRS

Move System Register allows the PE to read an AArch64 System register into a general-purpose register.



System variant

```
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 D7 *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

```
X[t] = AArch64.SysRegRead(sys_op0, sys_op1, sys_crn, sys_crm, sys_op2);
```

C6.2.163 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 are D, A, I, F, and SP. This set of bits is expanded in extensions to the architecture as follows:

- ARMv8.1 adds the PAN bit.
- ARMv8.2 adds the UAO bit.

31	30	29	28	27	26	25	24	23	22	21	20	19	18 16	15	14	13	12	11 8	7 5	4	3	2	1	0
1	1	0	1	0	1	0	1	0	0	0	0	0	op1	0	1	0	0	CRm	op2	1	1	1	1	1

System variant

```
MSR <pstatefield>, #<imm>
```

Decode for this encoding

```
AArch64.CheckSystemAccess('00', op1, '0100', CRm, op2, '11111', '0');

bits(4) operand = CRm;
PSTATEField field;
case op1:op2 of
    when '000 011'
        if !HaveUAOExt() then
            UnallocatedEncoding();
        field = PSTATEField_UAO;
    when '000 100'
        if !HavePANExt() then
            UnallocatedEncoding();
        field = PSTATEField_PAN;
    when '000 101' field = PSTATEField_SP;
    when '011 110' field = PSTATEField_DAIFSet;
    when '011 111' field = PSTATEField_DAIFClr;
    otherwise UnallocatedEncoding();

// Check that an AArch64 MSR/MRS access to the DAIF flags is permitted
if op1 == '011' && PSTATE.EL == EL0 && (IsInHost() || SCTLR_EL1.UMA == '0') then
    AArch64.SystemRegisterTrap(EL1, '00', op2, op1, '0100', '11111', CRm, '0');
```

Assembler symbols

`<pstatefield>` Is a PSTATE field name, encoded in the "op1:op2" field. It can have the following values:

SPSel	when op1 = 000, op2 = 101
DAIFSet	when op1 = 011, op2 = 110
DAIFClr	when op1 = 011, op2 = 111

When ARMv8.2-UAO is implemented, the following value is also valid:

UAO	when op1 = 000, op2 = 011

When ARMv8.1-PAN is implemented, the following value is also valid:

PAN	when op1 = 000, op2 = 100

The following encodings are reserved:

- op1 = 000, op2 = 00x.

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

- `op1 = 000`, `op2 = 010`.
- `op1 = 000`, `op2 = 11x`.
- `op1 = 001`, `op2 = xxx`.
- `op1 = 010`, `op2 = xxx`.
- `op1 = 011`, `op2 = 0xx`.
- `op1 = 011`, `op2 = 10x`.
- `op1 = 1xx`, `op2 = xxx`.

`<imm>`	Is a 4-bit unsigned immediate, in the range 0 to 15, encoded in the "CRm" field.

Operation

```
case field of
    when PSTATEField_SP
        PSTATE.SP = operand<0>;
    when PSTATEField_DAIFSet
        PSTATE.D = PSTATE.D OR operand<3>;
        PSTATE.A = PSTATE.A OR operand<2>;
        PSTATE.I = PSTATE.I OR operand<1>;
        PSTATE.F = PSTATE.F OR operand<0>;
    when PSTATEField_DAIFClr
        PSTATE.D = PSTATE.D AND NOT(operand<3>);
        PSTATE.A = PSTATE.A AND NOT(operand<2>);
        PSTATE.I = PSTATE.I AND NOT(operand<1>);
        PSTATE.F = PSTATE.F AND NOT(operand<0>);
    when PSTATEField_PAN
        PSTATE.PAN = operand<0>;
    when PSTATEField_UAO
        PSTATE.UAO = operand<0>;
```

C6.2.164 MSR (register)

Move general-purpose register to System Register allows the PE to write an AArch64 System register from a general-purpose register.

31	30	29	28	27	26	25	24	23	22	21	20	19	18 16	15 12	11 8	7 5	4 0
1	1	0	1	0	1	0	1	0	0	0	1	o0	op1	CRn	CRm	op2	Rt
										L							

System variant

```
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 D7 *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, X[t]);
```

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.165 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.

31	30 29	28 27 26 25 24	23 22 21	20 … 16	15	14 … 10	9 … 5	4 … 0
sf	0 0	1 1 0 1 1	0 0 0	Rm	1	Ra	Rn	Rd
					o0			

32-bit 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);
integer destsize = if sf == '1' then 64 else 32;
```

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];
bits(destsize) operand2 = X[m];
bits(destsize) operand3 = X[a];

integer result;

result = UInt(operand3) - (UInt(operand1) * UInt(operand2));
X[d] = result<destsize-1:0>;
```

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.166 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 for this instruction.

31	30	29	28	27	26	25	24	23	22	21	20 … 16	15	14	13	12	11	10	9 … 5	4 … 0
sf	0	0	1	1	0	1	1	0	0	0	Rm	0	1	1	1	1	1	Rn	Rd
												o0			Ra				

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.

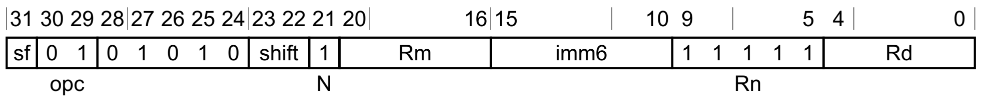
Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.167 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 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,

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

The description of ORN (shifted register) gives the operational pseudocode for this instruction.

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.168 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 for this instruction.

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

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.

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

The description of SUB (shifted register) gives the operational pseudocode for this instruction.

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.169 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 for this instruction.

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

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.

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

The description of SUBS (shifted register) gives the operational pseudocode for this instruction.

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.170 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 for this instruction.

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

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.

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.171 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 for this instruction.

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

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.

C6.2.172 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.

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

System variant

```
NOP
```

Decode for this encoding

```
// Empty.
```

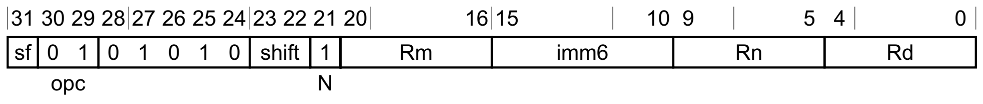
Operation

```
// do nothing
```

C6.2.173 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);
integer datasize = if sf == '1' then 64 else 32;
if sf == '0' && imm6<5> == '1' then ReservedValue();

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

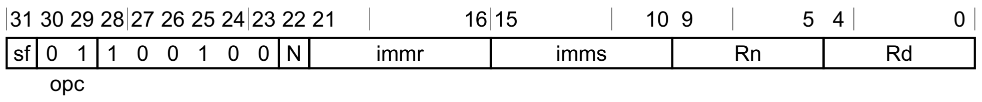
```
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = ShiftReg(m, shift_type, shift_amount);

operand2 = NOT(operand2);

result = operand1 OR operand2;
X[d] = result;
```

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.174 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);
integer datasize = if sf == '1' then 64 else 32;
bits(datasize) imm;
if sf == '0' && N != '0' then ReservedValue();
(imm, -) = DecodeBitMasks(N, imms, immr, TRUE);
```

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];

result = operand1 OR imm;
if d == 31 then
    SP[] = result;
else
    X[d] = result;
```

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.175 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.

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

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);
integer datasize = if sf == '1' then 64 else 32;
if sf == '0' && imm6<5> == '1' then ReservedValue();

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`

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

`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];
bits(datasize) operand2 = ShiftReg(m, shift_type, shift_amount);

result = operand1 OR operand2;
X[d] = result;
```

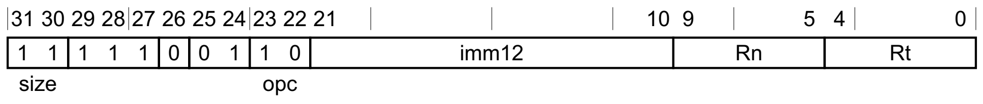
Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.176 PRFM (immediate)

Prefetch Memory (immediate) signals the memory system that data memory accesses from a specified address are likely to occur in the near future. The memory system can respond by taking actions that are expected to speed up the memory accesses when they do occur, such as preloading the cache line containing the specified address into one or more caches.

The effect of an `PRFM` instruction is IMPLEMENTATION DEFINED. For more information, see *Prefetch memory* on page C3-171.

For information about memory accesses, see *Load/Store addressing modes* on page C1-143.



Unsigned offset variant

```
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`.

`<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* on page C3-171.

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;

if n == 31 then
    address = SP[];
else
    address = X[n];

address = address + offset;

Prefetch(address, t<4:0>);
```

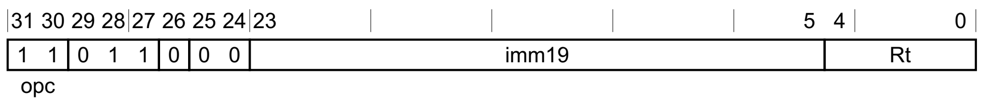
Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.177 PRFM (literal)

Prefetch Memory (literal) signals the memory system that data memory accesses from a specified address are likely to occur in the near future. The memory system can respond by taking actions that are expected to speed up the memory accesses when they do occur, such as preloading the cache line containing the specified address into one or more caches.

The effect of an PRFM instruction is IMPLEMENTATION DEFINED. For more information, see *Prefetch memory* on page C3-171.

For information about memory accesses, see *Load/Store addressing modes* on page C1-143.



Literal variant

```
PRFM (<prfop>|#<imm5>), <label>
```

Decode for this encoding

```
integer t = UInt(Rt);
bits(64) offset;

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

Assembler symbols

`<prfop>`	Is the prefetch operation, defined as `<type><target><policy>`.

`<type>` is one of:

PLD	Prefetch for load, encoded in the "Rt<4:3>" field as `0b00`.
PLI	Preload instructions, encoded in the "Rt<4:3>" field as `0b01`.
PST	Prefetch for store, encoded in the "Rt<4:3>" field as `0b10`.

`<target>` is one of:

L1	Level 1 cache, encoded in the "Rt<2:1>" field as `0b00`.
L2	Level 2 cache, encoded in the "Rt<2:1>" field as `0b01`.
L3	Level 3 cache, encoded in the "Rt<2:1>" field as `0b10`.

`<policy>` is one of:

KEEP	Retained or temporal prefetch, allocated in the cache normally. Encoded in the "Rt<0>" field as `0`.
STRM	Streaming or non-temporal prefetch, for data that is used only once. Encoded in the "Rt<0>" field as 1.

For more information on these prefetch operations, see *Prefetch memory* on page C3-171.

For other encodings of the "Rt" field, use `<imm5>`.

`<imm5>`	Is the prefetch operation encoding as an immediate, in the range 0 to 31, encoded in the "Rt" field. This syntax is only for encodings that are not accessible using `<prfop>`.
`<label>`	Is the program label from which the data is to be loaded. Its offset from the address of this instruction, in the range +/-1MB, is encoded as "imm19" times 4.

Operation

```
bits(64) address = PC[] + offset;

Prefetch(address, t<4:0>);
```

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.178 PRFM (register)

Prefetch Memory (register) signals the memory system that data memory accesses from a specified address are likely to occur in the near future. The memory system can respond by taking actions that are expected to speed up the memory accesses when they do occur, such as preloading the cache line containing the specified address into one or more caches.

The effect of an PRFM instruction is IMPLEMENTATION DEFINED. For more information, see *Prefetch memory* on page C3-171.

For information about memory accesses, see *Load/Store addressing modes* on page C1-143.

31 30	29 28 27	26	25 24	23 22	21	20 … 16	15 … 13	12	11 10	9 … 5	4 … 0
1 1	1 1 1	0	0 0	1 0	1	Rm	option	S	1 0	Rn	Rt
size				opc							

Integer variant

```
PRFM (<prfop>|#<imm5>), [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

Decode for this encoding

```
if option<1> == '0' then UnallocatedEncoding();    // 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.

<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* on page C3-171.

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);
bits(64) address;

if n == 31 then
    address = SP[];
else
    address = X[n];

address = address + offset;

Prefetch(address, t<4:0>);
```

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.179 PRFM (unscaled offset)

Prefetch Memory (unscaled offset) signals the memory system that data memory accesses from a specified address are likely to occur in the near future. The memory system can respond by taking actions that are expected to speed up the memory accesses when they do occur, such as preloading the cache line containing the specified address into one or more caches.

The effect of an `PRFUM` instruction is IMPLEMENTATION DEFINED. For more information, see *Prefetch memory* on page C3-171.

For information about memory accesses, see *Load/Store addressing modes* on page C1-143.

31 30	29 28 27	26	25 24	23 22	21	20 … 12	11 10	9 … 5	4 … 0
1 1	1 1 1	0	0 0	1 0	0	imm9	0 0	Rn	Rt
size				opc					

Unscaled offset variant

```
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* on page C3-171.
	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;

if n == 31 then
    address = SP[];
else
    address = X[n];

address = address + offset;

Prefetch(address, t<4:0>);
```

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.180 PSB CSYNC

Profiling Synchronization Barrier. This instruction is a barrier that ensures that all existing profiling data for the current PE has been formatted, and profiling buffer addresses have been translated such that all writes to the profiling buffer have been initiated. A following `DSB` instruction completes when the writes to the profiling buffer have completed.

ARMv8.2

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

System variant

```
PSB CSYNC
```

Decode for this encoding

```
SystemHintOp op;

op = if HaveStatisticalProfiling() then SystemHintOp_PSB else SystemHintOp_NOP;
```

Operation

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

    when SystemHintOp_WFE
        if IsEventRegisterSet() then
            ClearEventRegister();
        else
            if PSTATE.EL == EL0 then
                // Check for traps described by the OS which may be EL1 or EL2.
                AArch64.CheckForWFxTrap(EL1, TRUE);
            if HaveEL(EL2) && !IsSecure() && PSTATE.EL IN {EL0, EL1} && !IsInHost() then
                // Check for traps described by the Hypervisor.
                AArch64.CheckForWFxTrap(EL2, TRUE);
            if HaveEL(EL3) && PSTATE.EL != EL3 then
                // Check for traps described by the Secure Monitor.
                AArch64.CheckForWFxTrap(EL3, TRUE);
            WaitForEvent();

    when SystemHintOp_WFI
        if !InterruptPending() then
            if PSTATE.EL == EL0 then
                // Check for traps described by the OS which may be EL1 or EL2.
                AArch64.CheckForWFxTrap(EL1, FALSE);
            if HaveEL(EL2) && !IsSecure() && PSTATE.EL IN {EL0, EL1} && !IsInHost() then
                // Check for traps described by the Hypervisor.
                AArch64.CheckForWFxTrap(EL2, FALSE);
            if HaveEL(EL3) && PSTATE.EL != EL3 then
                // Check for traps described by the Secure Monitor.
                AArch64.CheckForWFxTrap(EL3, FALSE);
            WaitForInterrupt();

    when SystemHintOp_SEV
        SendEvent();

    when SystemHintOp_SEVL
        SendEventLocal();
```

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

```
when SystemHintOp_ESB
    ErrorSynchronizationBarrier(MBReqDomain_FullSystem, MBReqTypes_All);
    AArch64.ESBOperation();
    if HaveEL(EL2) && !IsSecure() && PSTATE.EL IN {EL0, EL1} then AArch64.vESBOperation();
    TakeUnmaskedSErrorInterrupts();

when SystemHintOp_PSB
    ProfilingSynchronizationBarrier();

otherwise    // do nothing
```

C6.2.181 RBIT

Reverse Bits reverses the bit order in a register.

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

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);

integer datasize = if sf == '1' then 64 else 32;
```

Assembler symbols

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

Operation

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

for i = 0 to datasize-1
    result<datasize-1-i> = operand<i>;

X[d] = result;
```

C6.2.182 RET

Return from subroutine branches unconditionally to an address in a register, with a hint that this is a subroutine return.

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

Integer variant

```
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];

BranchTo(target, BranchType_RET);
```

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.183 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.

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

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

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

integer datasize = if sf == '1' then 64 else 32;

integer container_size;
case opc of
    when '00'
        Unreachable();
    when '01'
        container_size = 16;
    when '10'
        container_size = 32;
    when '11'
        if sf == '0' then UnallocatedEncoding();
        container_size = 64;
```

Assembler symbols

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

Operation

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

integer containers = datasize DIV container_size;
integer elements_per_container = container_size DIV 8;
integer index = 0;
integer rev_index;
```

```
for c = 0 to containers-1
    rev_index = index + ((elements_per_container - 1) * 8);
    for e = 0 to elements_per_container-1
        result<rev_index+7:rev_index> = operand<index+7:index>;
        index = index + 8;
        rev_index = rev_index - 8;

X[d] = result;
```

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.184 REV16

Reverse bytes in 16-bit halfwords reverses the byte order in each 16-bit halfword of a register.

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

32-bit 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

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

integer datasize = if sf == '1' then 64 else 32;

integer container_size;
case opc of
    when '00'
        Unreachable();
    when '01'
        container_size = 16;
    when '10'
        container_size = 32;
    when '11'
        if sf == '0' then UnallocatedEncoding();
        container_size = 64;
```

Assembler symbols

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

Operation

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

integer containers = datasize DIV container_size;
integer elements_per_container = container_size DIV 8;
integer index = 0;
integer rev_index;
for c = 0 to containers-1
    rev_index = index + ((elements_per_container - 1) * 8);
    for e = 0 to elements_per_container-1
```

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

```
        result<rev_index+7:rev_index> = operand<index+7:index>;
        index = index + 8;
        rev_index = rev_index - 8;

X[d] = result;
```

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.185 REV32

Reverse bytes in 32-bit words reverses the byte order in each 32-bit word of a register.

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

64-bit variant

```
REV32 <Xd>, <Xn>
```

Decode for this encoding

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

integer datasize = if sf == '1' then 64 else 32;

integer container_size;
case opc of
    when '00'
        Unreachable();
    when '01'
        container_size = 16;
    when '10'
        container_size = 32;
    when '11'
        if sf == '0' then UnallocatedEncoding();
        container_size = 64;
```

Assembler symbols

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

Operation

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

integer containers = datasize DIV container_size;
integer elements_per_container = container_size DIV 8;
integer index = 0;
integer rev_index;
for c = 0 to containers-1
    rev_index = index + ((elements_per_container - 1) * 8);
    for e = 0 to elements_per_container-1
        result<rev_index+7:rev_index> = operand<index+7:index>;
        index = index + 8;
        rev_index = rev_index - 8;

X[d] = result;
```

C6.2.186 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 for this instruction.

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

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.

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.187 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 for this instruction.

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

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.

C6.2.188 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 for this instruction.

31	30	29	28	27	26	25	24	23	22	21	20-16	15	14	13	12	11	10	9-5	4-0
sf	0	0	1	1	0	1	0	1	1	0	Rm	0	0	1	0	1	1	Rn	Rd
																op2			

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.

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.189 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.

31	30	29	28	27	26	25	24	23	22	21	20 16	15	14	13	12	11	10	9 5	4 0
sf	0	0	1	1	0	1	0	1	1	0	Rm	0	0	1	0	1	1	Rn	Rd
																op2			

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);
integer datasize = if sf == '1' then 64 else 32;
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

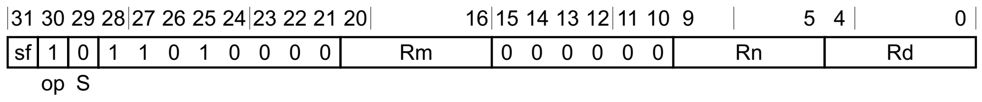
```
bits(datasize) result;
bits(datasize) operand2 = X[m];

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

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.190 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);
integer datasize = if sf == '1' then 64 else 32;
```

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];
bits(datasize) operand2 = X[m];

operand2 = NOT(operand2);
```

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

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

X[d] = result;
```

C6.2.191 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.

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

32-bit 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);
integer datasize = if sf == '1' then 64 else 32;
```

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];
bits(datasize) operand2 = X[m];
bits(4) nzcv;

operand2 = NOT(operand2);
```

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

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

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

X[d] = result;
```

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.192 SBFIZ

Signed Bitfield Insert in Zero zeroes the destination register and copies any number of contiguous bits from a source register into any position in the destination register, sign-extending the most significant bit of the transferred value.

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 for this instruction.

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

32-bit 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.

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.193 SBFM

Signed Bitfield Move copies any number of low-order bits from a source register into the same number of adjacent bits at any position in the destination register, shifting in copies of the sign bit in the upper bits and zeros in the lower bits.

This instruction is used by the aliases ASR (immediate), SBFIZ, SBFX, SXTB, SXTH, and SXTW. See *Alias conditions* on page C6-821 for details of when each alias is preferred.

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

32-bit 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);
integer datasize = if sf == '1' then 64 else 32;

integer R;
integer S;
bits(datasize) wmask;
bits(datasize) tmask;

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

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

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

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];

// 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>);

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

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.194 SBFX

Signed Bitfield Extract extracts any number of adjacent bits at any position from a register, sign-extends them 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 for this instruction.

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

32-bit 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.

C6.2.195 SDIV

Signed Divide divides a signed integer register value by another signed integer register value, and writes the result to the destination register. The condition flags are not affected.

31	30	29	28	27	26	25	24	23	22	21	20 16	15	14	13	12	11	10	9 5	4 0
sf	0	0	1	1	0	1	0	1	1	0	Rm	0	0	0	0	1	1	Rn	Rd
																	o1		

32-bit 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);
integer datasize = if sf == '1' then 64 else 32;
```

Assembler symbols

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

Operation

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

if IsZero(operand2) then
    result = 0;
else
    result = RoundTowardsZero(Real(Int(operand1, FALSE)) / Real(Int(operand2, FALSE)));

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

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.196 SEV

Send Event is a hint instruction. It causes an event to be signaled to all PEs in the multiprocessor system. For more information, see *Wait for Event mechanism and Send event* on page D1-1879.

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

System variant

```
SEV
```

Decode for this encoding

```
// Empty.
```

Operation

```
SendEvent();
```

C6.2.197 SEVL

Send Event Local is a hint instruction that causes an event to be signaled locally without requiring the event to be signaled to other PEs in the multiprocessor system. It can prime a wait-loop which starts with a `WFE` instruction.

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

System variant

```
SEVL
```

Decode for this encoding

```
// Empty.
```

Operation

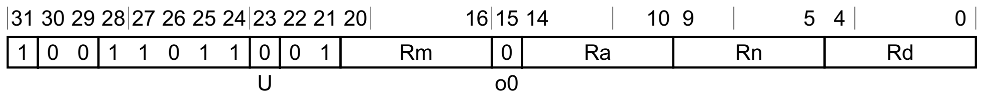
```
SendEventLocal();
```

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.198 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.



64-bit variant

```
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];
bits(32) operand2 = X[m];
bits(64) operand3 = X[a];

integer result;

result = Int(operand3, FALSE) + (Int(operand1, FALSE) * Int(operand2, FALSE));

X[d] = result<63:0>;
```

C6.2.199 SMC

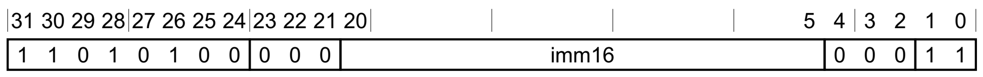
Secure Monitor Call causes an exception to EL3.

SMC is available only for software executing at EL1 or higher. It is UNDEFINED in EL0.

If the values of HCR_EL2.TSC and SCR_EL3.SMD are both 0, execution of an SMC instruction at EL1 or higher generates a Secure Monitor Call exception, recording it in ESR_ELx, using the EC value 0x17, that is taken to EL3.

If the value of HCR_EL2.TSC is 1, execution of an SMC instruction in a Non-secure EL1 state generates an exception that is taken to EL2, regardless of the value of SCR_EL3.SMD. For more information, see *Traps to EL2 of Non-secure EL1 execution of SMC instructions* on page D1-1858.

If the value of HCR_EL2.TSC is 0 and the value of SCR_EL3.SMD is 1, the SMC instruction is UNDEFINED.



System variant

```
SMC #<imm>
```

Decode for this encoding

```
bits(16) imm = imm16;
```

Assembler symbols

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

Operation

```
if !HaveEL(EL3) || PSTATE.EL == EL0 then
    UnallocatedEncoding();

AArch64.CheckForSMCTrap(imm);

if SCR_EL3.SMD == '1' then
    // SMC disabled
    AArch64.UndefinedFault();
else
    AArch64.CallSecureMonitor(imm);
```

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.200 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 for this instruction.

31	30 29	28 27 26 25 24	23	22 21	20 16	15	14 10	9 5	4 0
1	0 0	1 1 0 1 1	0	0 1	Rm	1	1 1 1 1 1	Rn	Rd
			U			o0	Ra		

64-bit variant

```
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.

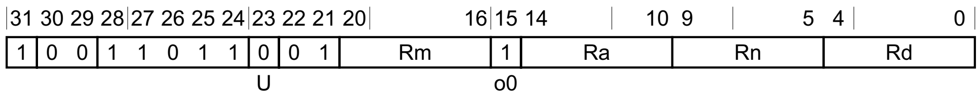
Operation

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

C6.2.201 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.



64-bit variant

```
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.

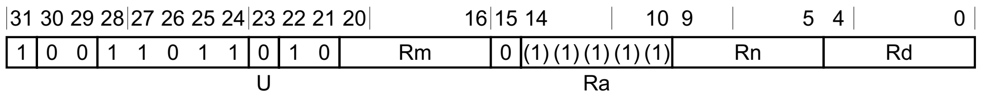
Operation

```
bits(32) operand1 = X[n];
bits(32) operand2 = X[m];
bits(64) operand3 = X[a];

integer result;

result = Int(operand3, FALSE) - (Int(operand1, FALSE) * Int(operand2, FALSE));
X[d] = result<63:0>;
```

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.202 SMULH

Signed Multiply High multiplies two 64-bit register values, and writes bits[127:64] of the 128-bit result to the 64-bit destination register.



64-bit variant

```
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];
bits(64) operand2 = X[m];

integer result;

result = Int(operand1, FALSE) * Int(operand2, FALSE);

X[d] = result<127:64>;
```

C6.2.203 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 for this instruction.

31	30 29	28 27 26 25 24	23	22 21	20 16	15	14 10	9 5	4 0
1	0 0	1 1 0 1 1	0	0 1	Rm	0	1 1 1 1 1	Rn	Rd
			U			o0	Ra		

64-bit variant

```
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.

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.204 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` has no memory ordering semantics.
- `STADDLB` stores to memory with release semantics, as described in *Load-Acquire, Store-Release* on page B2-94.

For information about memory accesses see *Load/Store addressing modes* on page C1-143.

ARMv8.1

31 30	29 28 27	26	25 24	23	22	21	20 16	15	14 12	11 10	9 5	4 3 2 1 0
0 0	1 1 1	0	0 0	0	R	1	Rs	0	0 0 0	0 0	Rn	1 1 1 1 1
size		V		A				o3	opc			

No memory ordering variant

Applies when `R == 0`.

```
STADDB <Ws>, [<Xn|SP>]
```

Release variant

Applies when `R == 1`.

```
STADDLB <Ws>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !HaveAtomicExt() then UnallocatedEncoding();
integer n = UInt(Rn);
integer s = UInt(Rs);

AccType stacctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
```

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

```
bits(64) address;
bits(8) value;
bits(8) data;
bits(8) result;

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, 1, AccType_ATOMICRW];

result = data + value;
```

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

```
// All observers in the shareability domain observe the
// following load and store atomically.
Mem[address, 1, stacctype] = result;
```

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.205 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` has no memory ordering semantics.
- `STADDLH` stores to memory with release semantics, as described in *Load-Acquire, Store-Release* on page B2-94.

For information about memory accesses see *Load/Store addressing modes* on page C1-143.

ARMv8.1

31 30	29 28 27	26	25 24	23	22	21	20 16	15	14 12	11 10	9 5	4 3 2 1 0
0 1	1 1 1	0	0 0	0	R	1	Rs	0	0 0 0	0 0	Rn	1 1 1 1 1
size		V		A				o3	opc			

No memory ordering variant

Applies when `R == 0`.

```
STADDH <Ws>, [<Xn|SP>]
```

Release variant

Applies when `R == 1`.

```
STADDLH <Ws>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !HaveAtomicExt() then UnallocatedEncoding();
integer n = UInt(Rn);
integer s = UInt(Rs);

AccType stacctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
```

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

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

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, 2, AccType_ATOMICRW];

result = data + value;
```

```
// All observers in the shareability domain observe the
// following load and store atomically.
Mem[address, 2, stacctype] = result;
```

C6.2.206 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` has no memory ordering semantics.
- `STADDL` stores to memory with release semantics, as described in *Load-Acquire, Store-Release* on page B2-94.

For information about memory accesses see *Load/Store addressing modes* on page C1-143.

ARMv8.1

31 30	29 28 27	26	25 24	23	22	21	20 16	15	14 12	11 10	9 5	4 3 2 1 0
1 x	1 1 1	0	0 0	0	R	1	Rs	0	0 0 0	0 0	Rn	1 1 1 1 1
size		V		A				o3	opc			

32-bit, no memory ordering variant

Applies when `size == 10 && R == 0`.

`STADD <Ws>, [<Xn|SP>]`

32-bit, release variant

Applies when `size == 10 && R == 1`.

`STADDL <Ws>, [<Xn|SP>]`

64-bit, no memory ordering variant

Applies when `size == 11 && R == 0`.

`STADD <Xs>, [<Xn|SP>]`

64-bit, release variant

Applies when `size == 11 && R == 1`.

`STADDL <Xs>, [<Xn|SP>]`

Decode for all variants of this encoding

```
if !HaveAtomicExt() then UnallocatedEncoding();
integer n = UInt(Rn);
integer s = UInt(Rs);

integer datasize = 8 << UInt(size);
AccType stacctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
```

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.

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

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

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, datasize DIV 8, AccType_ATOMICRW];

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

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.207 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` has no memory ordering semantics.
- `STCLRLB` stores to memory with release semantics, as described in *Load-Acquire, Store-Release* on page B2-94.

For information about memory accesses see *Load/Store addressing modes* on page C1-143.

ARMv8.1

31	30	29	28	27	26	25	24	23	22	21	20 16	15	14 12	11	10	9 5	4 3 2 1 0
0	0	1	1	1	0	0	0	0	R	1	Rs	0	0 0 1	0	0	Rn	1 1 1 1 1
size					V			A				o3	opc				

No memory ordering variant

Applies when `R == 0`.

```
STCLRB <Ws>, [<Xn|SP>]
```

Release variant

Applies when `R == 1`.

```
STCLRLB <Ws>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !HaveAtomicExt() then UnallocatedEncoding();
integer n = UInt(Rn);
integer s = UInt(Rs);

AccType stacctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
```

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

```
bits(64) address;
bits(8) value;
bits(8) data;
bits(8) result;

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, 1, AccType_ATOMICRW];

result = data AND NOT(value);
```

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

```
// All observers in the shareability domain observe the
// following load and store atomically.
Mem[address, 1, stacctype] = result;
```

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.208 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` has no memory ordering semantics.
- `STCLRLH` stores to memory with release semantics, as described in *Load-Acquire, Store-Release* on page B2-94.

For information about memory accesses see *Load/Store addressing modes* on page C1-143.

ARMv8.1

31 30	29 28 27	26	25 24	23	22	21	20 ... 16	15	14 ... 12	11 10	9 ... 5	4 3 2 1 0
0 1	1 1 1	0	0 0	0	R	1	Rs	0	0 0 1	0 0	Rn	1 1 1 1 1
size		V		A				o3	opc			

No memory ordering variant

Applies when `R == 0`.

`STCLRH <Ws>, [<Xn|SP>]`

Release variant

Applies when `R == 1`.

`STCLRLH <Ws>, [<Xn|SP>]`

Decode for all variants of this encoding

```
if !HaveAtomicExt() then UnallocatedEncoding();
integer n = UInt(Rn);
integer s = UInt(Rs);

AccType stacctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
```

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

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

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, 2, AccType_ATOMICRW];

result = data AND NOT(value);
```

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

```
// All observers in the shareability domain observe the
// following load and store atomically.
Mem[address, 2, stacctype] = result;
```

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.209 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` has no memory ordering semantics.
- `STCLRL` stores to memory with release semantics, as described in *Load-Acquire, Store-Release* on page B2-94.

For information about memory accesses see *Load/Store addressing modes* on page C1-143.

ARMv8.1

31 30	29 28 27	26	25 24	23	22	21	20 16	15	14 12	11 10	9 5	4 3 2 1 0
1 x	1 1 1	0	0 0	0	R	1	Rs	0	0 0 1	0 0	Rn	1 1 1 1 1
size		V		A				o3	opc			

32-bit, no memory ordering variant

Applies when `size == 10 && R == 0`.

```
STCLR <Ws>, [<Xn|SP>]
```

32-bit, release variant

Applies when `size == 10 && R == 1`.

```
STCLRL <Ws>, [<Xn|SP>]
```

64-bit, no memory ordering variant

Applies when `size == 11 && R == 0`.

```
STCLR <Xs>, [<Xn|SP>]
```

64-bit, release variant

Applies when `size == 11 && R == 1`.

```
STCLRL <Xs>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !HaveAtomicExt() then UnallocatedEncoding();
integer n = UInt(Rn);
integer s = UInt(Rs);

integer datasize = 8 << UInt(size);
AccType stacctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
```

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.

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

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

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, datasize DIV 8, AccType_ATOMICRW];

result = data AND NOT(value);
// All observers in the shareability domain observe the
// following load and store atomically.
Mem[address, datasize DIV 8, stacctype] = result;
```

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.210 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` has no memory ordering semantics.
- `STEORLB` stores to memory with release semantics, as described in *Load-Acquire, Store-Release* on page B2-94.

For information about memory accesses see *Load/Store addressing modes* on page C1-143.

ARMv8.1

31 30	29 28 27	26	25 24	23	22	21	20 16	15	14 12	11 10	9 5	4 3 2 1 0
0 0	1 1 1	0	0 0	0	R	1	Rs	0	0 1 0	0 0	Rn	1 1 1 1 1
size		V		A				o3	opc			

No memory ordering variant

Applies when `R == 0`.

```
STEORB <Ws>, [<Xn|SP>]
```

Release variant

Applies when `R == 1`.

```
STEORLB <Ws>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !HaveAtomicExt() then UnallocatedEncoding();
integer n = UInt(Rn);
integer s = UInt(Rs);

AccType stacctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
```

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

```
bits(64) address;
bits(8) value;
bits(8) data;
bits(8) result;

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, 1, AccType_ATOMICRW];

result = data EOR value;
```

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

```
// All observers in the shareability domain observe the
// following load and store atomically.
Mem[address, 1, stacctype] = result;
```

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.211 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 has no memory ordering semantics.
- STEORLH stores to memory with release semantics, as described in *Load-Acquire, Store-Release* on page B2-94.

For information about memory accesses see *Load/Store addressing modes* on page C1-143.

ARMv8.1

31 30	29 28 27	26	25 24	23	22	21	20 16	15	14 12	11 10	9 5	4 3 2 1 0
0 1	1 1 1	0	0 0	0	R	1	Rs	0	0 1 0	0 0	Rn	1 1 1 1 1
size		V		A				o3	opc			

No memory ordering variant

Applies when `R == 0`.

```
STEORH <Ws>, [<Xn|SP>]
```

Release variant

Applies when `R == 1`.

```
STEORLH <Ws>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !HaveAtomicExt() then UnallocatedEncoding();
integer n = UInt(Rn);
integer s = UInt(Rs);

AccType stacctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
```

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

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

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, 2, AccType_ATOMICRW];

result = data EOR value;
```

```
// All observers in the shareability domain observe the
// following load and store atomically.
Mem[address, 2, stacctype] = result;
```

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.212 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` has no memory ordering semantics.
- `STEORL` stores to memory with release semantics, as described in *Load-Acquire, Store-Release* on page B2-94.

For information about memory accesses see *Load/Store addressing modes* on page C1-143.

ARMv8.1

31 30	29 28 27	26	25 24	23	22	21	20 16	15	14 12	11 10	9 5	4 3 2 1 0
1 x	1 1 1	0	0 0	0	R	1	Rs	0	0 1 0	0 0	Rn	1 1 1 1 1
size		V		A				o3	opc			

32-bit, no memory ordering variant

Applies when `size == 10 && R == 0`.

```
STEOR <Ws>, [<Xn|SP>]
```

32-bit, release variant

Applies when `size == 10 && R == 1`.

```
STEORL <Ws>, [<Xn|SP>]
```

64-bit, no memory ordering variant

Applies when `size == 11 && R == 0`.

```
STEOR <Xs>, [<Xn|SP>]
```

64-bit, release variant

Applies when `size == 11 && R == 1`.

```
STEORL <Xs>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !HaveAtomicExt() then UnallocatedEncoding();
integer n = UInt(Rn);
integer s = UInt(Rs);

integer datasize = 8 << UInt(size);
AccType stacctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
```

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

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

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, datasize DIV 8, AccType_ATOMICRW];

result = data EOR value;
// All observers in the shareability domain observe the
// following load and store atomically.
Mem[address, datasize DIV 8, stacctype] = result;
```

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.213 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* on page B2-95. For information about memory accesses, see *Load/Store addressing modes* on page C1-143.

ARMv8.1

31 30	29 28 27 26 25 24	23	22	21	20 16	15	14 10	9 5	4 0
0 0	0 0 1 0 0 0	1	0	0	(1) (1) (1) (1) (1)	0	(1) (1) (1) (1) (1)	Rn	Rt
size			L		Rs	o0	Rt2		

No offset variant

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

Decode for this encoding

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

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;

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

data = X[t];
Mem[address, 1, AccType_LIMITEDORDERED] = data;
```

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.214 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* on page B2-95. For information about memory accesses, see *Load/Store addressing modes* on page C1-143.

ARMv8.1

31 30	29 28 27 26 25 24	23	22	21	20 16	15	14 10	9 5	4 0
0 1	0 0 1 0 0 0	1	0	0	(1) (1) (1) (1) (1)	0	(1) (1) (1) (1) (1)	Rn	Rt
size			L		Rs	o0	Rt2		

No offset variant

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

Decode for this encoding

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

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;

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

data = X[t];
Mem[address, 2, AccType_LIMITEDORDERED] = data;
```

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.215 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* on page B2-95. For information about memory accesses, see *Load/Store addressing modes* on page C1-143.

ARMv8.1

31	30	29	28	27	26	25	24	23	22	21	20				16	15	14				10	9				5	4				0
1	x	0	0	1	0	0	0	1	0	0	(1)	(1)	(1)	(1)	(1)	0	(1)	(1)	(1)	(1)	(1)	Rn					Rt				
size									L		Rs					o0	Rt2														

32-bit 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);

integer elsize = 8 << UInt(size);
```

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;

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

data = X[t];
Mem[address, dbytes, AccType_LIMITEDORDERED] = data;
```

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.216 STLR

Store-Release Register stores a 32-bit word or a 64-bit doubleword to a memory location, from a register. The instruction also has memory ordering semantics as described in *Load-Acquire, Store-Release* on page B2-94. For information about memory accesses, see *Load/Store addressing modes* on page C1-143.

31 30	29 28 27 26 25 24	23	22	21	20 … 16	15	14 … 10	9 … 5	4 … 0
1 x	0 0 1 0 0 0	1	0	0	(1) (1) (1) (1) (1)	1	(1) (1) (1) (1) (1)	Rn	Rt
size			L		Rs	o0	Rt2		

32-bit 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);

integer elsize = 8 << UInt(size);
```

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.

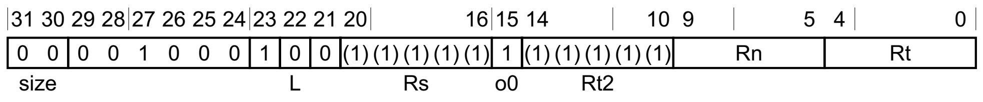
Operation

```
bits(64) address;
bits(elsize) data;
constant integer dbytes = elsize DIV 8;

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

data = X[t];
Mem[address, dbytes, AccType_ORDERED] = data;
```

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.217 STLRB

Store-Release Register Byte stores a byte from a 32-bit register to a memory location. The instruction also has memory ordering semantics as described in *Load-Acquire, Store-Release* on page B2-94. For information about memory accesses, see *Load/Store addressing modes* on page C1-143.


31 30 29 28 27 26 25 24 23 22 21 20 16 15 14 10 9 5 4 0
0 0 0 0 1 0 0 0 1 0 0 (1) (1) (1) (1) (1) 1 (1) (1) (1) (1) (1) Rn Rt
size
L
Rs
o0
Rt2

No offset variant

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

Decode for this encoding

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

Assembler symbols

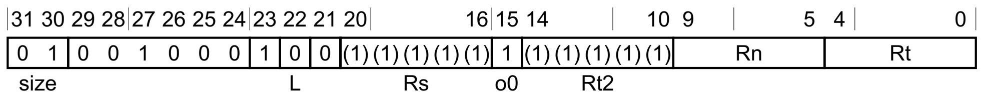
`<Wt>`	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
`<Xn|SP>`	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

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

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

data = X[t];
Mem[address, 1, AccType_ORDERED] = data;
```

C6.2.218 STLRH

Store-Release Register Halfword stores a halfword from a 32-bit register to a memory location. The instruction also has memory ordering semantics as described in *Load-Acquire, Store-Release* on page B2-94. For information about memory accesses, see *Load/Store addressing modes* on page C1-143.


31 30 29 28 27 26 25 24 23 22 21 20 16 15 14 10 9 5 4 0
0 1 0 0 1 0 0 0 1 0 0 (1) (1) (1) (1) (1) 1 (1) (1) (1) (1) (1) Rn Rt
size
L
Rs
o0
Rt2

No offset variant

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

Decode for this encoding

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

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;

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

data = X[t];
Mem[address, 2, AccType_ORDERED] = data;
```

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.219 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* on page B2-121. A 32-bit pair requires the address to be doubleword aligned and is single-copy atomic at doubleword granularity. A 64-bit pair requires the address to be quadword aligned and, if the Store-Exclusive succeeds, it causes a single-copy atomic update of the 128-bit memory location being updated. The instruction also has memory ordering semantics as described in *Load-Acquire, Store-Release* on page B2-94. For information about memory accesses see *Load/Store addressing modes* on page C1-143.

31	30	29	28	27	26	25	24	23	22	21	20 16	15	14 10	9 5	4 0
1	sz	0	0	1	0	0	0	0	0	1	Rs	1	Rt2	Rn	Rt
									L			o0			

32-bit 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

integer elsize = 32 << UInt(sz);
integer datasize = elsize * 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 *STLXP* on page K1-6129.

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.

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

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

Aborts and alignment

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- `<Ws>` is not updated.

Accessing an address that is not aligned to the size of the data being accessed causes an Alignment fault Data Abort exception to be generated, subject to the following rules:

- If `AArch64.ExclusiveMonitorsPass()` returns `TRUE`, the exception is generated.
- Otherwise, it is IMPLEMENTATION DEFINED whether the exception is generated.

If `AArch64.ExclusiveMonitorsPass()` returns `FALSE` and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.

Operation

```
bits(64) address;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;

if s == t || (s == t2) then
    Constraint c = ConstrainUnpredictable();
    assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN rt_unknown = TRUE;    // store UNKNOWN value
        when Constraint_NONE    rt_unknown = FALSE;    // store original value
        when Constraint_UNDEF   UnallocatedEncoding();
        when Constraint_NOP     EndOfInstruction();
if s == n && n != 31 then
    Constraint c = ConstrainUnpredictable();
    assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN rn_unknown = TRUE;    // address is UNKNOWN
        when Constraint_NONE    rn_unknown = FALSE;    // address is original base
        when Constraint_UNDEF   UnallocatedEncoding();
        when Constraint_NOP     EndOfInstruction();

if n == 31 then
    CheckSPAlignment();
    address = SP[];
elsif rn_unknown then
    address = bits(64) UNKNOWN;
else
    address = X[n];

if rt_unknown then
    data = bits(datasize) UNKNOWN;
else
    bits(datasize DIV 2) el1 = X[t];
    bits(datasize DIV 2) el2 = X[t2];
    data = if BigEndian() then el1:el2 else el2:el1;
bit status = '1';
// Check whether the Exclusive Monitors are set to include the
// physical memory locations corresponding to virtual address
// range [address, address+dbytes-1].
if AArch64.ExclusiveMonitorsPass(address, dbytes) then
    // This atomic write will be rejected if it does not refer
    // to the same physical locations after address translation.
```

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

```
        Mem[address, dbytes, AccType_ORDERED] = data;
        status = ExclusiveMonitorsStatus();
    X[s] = ZeroExtend(status, 32);
```

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.220 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* on page B2-121. The memory access is atomic. The instruction also has memory ordering semantics as described in *Load-Acquire, Store-Release* on page B2-94. For information about memory accesses see *Load/Store addressing modes* on page C1-143.

31 30	29 28 27 26 25 24	23	22	21	20 16	15	14 10	9 5	4 0
1 x	0 0 1 0 0 0	0	0	0	Rs	1	(1) (1) (1) (1) (1)	Rn	Rt
size			L			o0	Rt2		

32-bit 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

integer elsize = 8 << UInt(size);
```

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* on page K1-6129.

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.

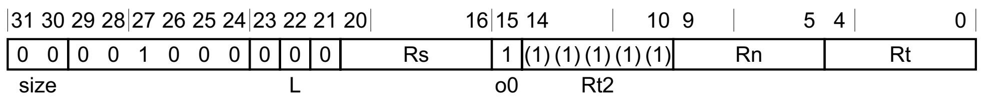
If `AArch64.ExclusiveMonitorsPass()` returns `FALSE` and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.

Operation

```
bits(64) address;
bits(elsize) data;
constant integer dbytes = elsize DIV 8;
boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;

if s == t then
    Constraint c = ConstrainUnpredictable();
    assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN rt_unknown = TRUE;    // store UNKNOWN value
        when Constraint_NONE    rt_unknown = FALSE;    // store original value
        when Constraint_UNDEF   UnallocatedEncoding();
        when Constraint_NOP     EndOfInstruction();
if s == n && n != 31 then
    Constraint c = ConstrainUnpredictable();
    assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN rn_unknown = TRUE;    // address is UNKNOWN
        when Constraint_NONE    rn_unknown = FALSE;    // address is original base
        when Constraint_UNDEF   UnallocatedEncoding();
        when Constraint_NOP     EndOfInstruction();

if n == 31 then
    CheckSPAlignment();
    address = SP[];
elsif rn_unknown then
    address = bits(64) UNKNOWN;
else
    address = X[n];

if rt_unknown then
    data = bits(elsize) UNKNOWN;
else
    data = X[t];

bit status = '1';
// Check whether the Exclusive Monitors are set to include the
// physical memory locations corresponding to virtual address
// range [address, address+dbytes-1].
if AArch64.ExclusiveMonitorsPass(address, dbytes) then
    // This atomic write will be rejected if it does not refer
    // to the same physical locations after address translation.
    Mem[address, dbytes, AccType_ORDERED] = data;
    status = ExclusiveMonitorsStatus();
X[s] = ZeroExtend(status, 32);
```

C6.2.221 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* on page B2-121. The memory access is atomic. The instruction also has memory ordering semantics as described in *Load-Acquire, Store-Release* on page B2-94. For information about memory accesses see *Load/Store addressing modes* on page C1-143.



No offset variant

```
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
```

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* on page K1-6130.

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;
boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;

if s == t then
    Constraint c = ConstrainUnpredictable();
```

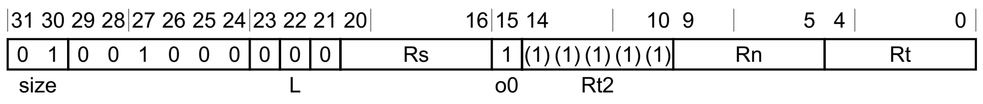
Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

```
    assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN rt_unknown = TRUE;    // store UNKNOWN value
        when Constraint_NONE    rt_unknown = FALSE;    // store original value
        when Constraint_UNDEF   UnallocatedEncoding();
        when Constraint_NOP     EndOfInstruction();
if s == n && n != 31 then
    Constraint c = ConstrainUnpredictable();
    assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN rn_unknown = TRUE;    // address is UNKNOWN
        when Constraint_NONE    rn_unknown = FALSE;    // address is original base
        when Constraint_UNDEF   UnallocatedEncoding();
        when Constraint_NOP     EndOfInstruction();

if n == 31 then
    CheckSPAlignment();
    address = SP[];
elsif rn_unknown then
    address = bits(64) UNKNOWN;
else
    address = X[n];

if rt_unknown then
    data = bits(8) UNKNOWN;
else
    data = X[t];

bit status = '1';
// Check whether the Exclusive Monitors are set to include the
// physical memory locations corresponding to virtual address
// range [address, address+dbytes-1].
if AArch64.ExclusiveMonitorsPass(address, 1) then
    // This atomic write will be rejected if it does not refer
    // to the same physical locations after address translation.
    Mem[address, 1, AccType_ORDERED] = data;
    status = ExclusiveMonitorsStatus();
X[s] = ZeroExtend(status, 32);
```

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.222 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* on page B2-121. The memory access is atomic. The instruction also has memory ordering semantics as described in *Load-Acquire, Store-Release* on page B2-94. For information about memory accesses see *Load/Store addressing modes* on page C1-143.



No offset variant

```
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
```

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* on page K1-6130.

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.

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
bits(64) address;
bits(16) data;
boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;

if s == t then
    Constraint c = ConstrainUnpredictable();
    assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN rt_unknown = TRUE;    // store UNKNOWN value
        when Constraint_NONE    rt_unknown = FALSE;    // store original value
        when Constraint_UNDEF   UnallocatedEncoding();
        when Constraint_NOP     EndOfInstruction();
if s == n && n != 31 then
    Constraint c = ConstrainUnpredictable();
    assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN rn_unknown = TRUE;    // address is UNKNOWN
        when Constraint_NONE    rn_unknown = FALSE;    // address is original base
        when Constraint_UNDEF   UnallocatedEncoding();
        when Constraint_NOP     EndOfInstruction();

if n == 31 then
    CheckSPAlignment();
    address = SP[];
elsif rn_unknown then
    address = bits(64) UNKNOWN;
else
    address = X[n];

if rt_unknown then
    data = bits(16) UNKNOWN;
else
    data = X[t];

bit status = '1';
// Check whether the Exclusive Monitors are set to include the
// physical memory locations corresponding to virtual address
// range [address, address+dbytes-1].
if AArch64.ExclusiveMonitorsPass(address, 2) then
    // This atomic write will be rejected if it does not refer
    // to the same physical locations after address translation.
    Mem[address, 2, AccType_ORDERED] = data;
    status = ExclusiveMonitorsStatus();
X[s] = ZeroExtend(status, 32);
```

C6.2.223 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* on page C1-143. For information about Non-temporal pair instructions, see *Load/Store Non-temporal Pair* on page C3-163.

31	30	29	28	27	26	25	24	23	22	21 15	14 10	9 5	4 0
x	0	1	0	1	0	0	0	0	0	imm7	Rt2	Rn	Rt
opc									L				

32-bit 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 UnallocatedEncoding();
integer scale = 2 + UInt(opc<1>);
integer datasize = 8 << scale;
bits(64) offset = LSL(SignExtend(imm7, 64), scale);
```

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
bits(64) address;
bits(datasize) data1;
bits(datasize) data2;
constant integer dbytes = datasize DIV 8;

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

address = address + offset;

data1 = X[t];
data2 = X[t2];
Mem[address, dbytes, AccType_STREAM] = data1;
Mem[address+dbytes, dbytes, AccType_STREAM] = data2;
```

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.224 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* on page C1-143.

Post-index

31	30	29	28	27	26	25	24	23	22	21 … 15	14 … 10	9 … 5	4 … 0
x	0	1	0	1	0	0	0	1	0	imm7	Rt2	Rn	Rt
opc									L				

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

31	30	29	28	27	26	25	24	23	22	21 … 15	14 … 10	9 … 5	4 … 0
x	0	1	0	1	0	0	1	1	0	imm7	Rt2	Rn	Rt
opc									L				

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

31	30	29	28	27	26	25	24	23	22	21 … 15	14 … 10	9 … 5	4 … 0
x	0	1	0	1	0	0	1	0	0	imm7	Rt2	Rn	Rt
opc									L				

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

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* on page K1-6129.

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 UnallocatedEncoding();
integer scale = 2 + UInt(opc<1>);
integer datasize = 8 << scale;
bits(64) offset = LSL(SignExtend(imm7, 64), scale);
```

Operation for all encodings

```
bits(64) address;
bits(datasize) data1;
bits(datasize) data2;
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;
```

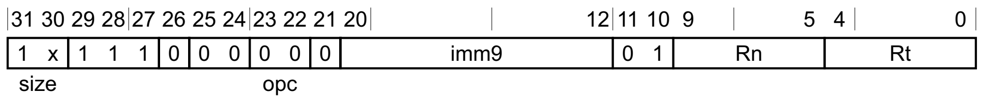
```
if wback && (t == n || t2 == n) && n != 31 then
    Constraint c = ConstrainUnpredictable();
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE    rt_unknown = FALSE;    // value stored is pre-writeback
        when Constraint_UNKNOWN rt_unknown = TRUE;    // value stored is UNKNOWN
        when Constraint_UNDEF   UnallocatedEncoding();
        when Constraint_NOP     EndOfInstruction();

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

if !postindex then
    address = address + offset;

if rt_unknown && t == n then
    data1 = bits(datasize) UNKNOWN;
else
    data1 = X[t];
if rt_unknown && t2 == n then
    data2 = bits(datasize) UNKNOWN;
else
    data2 = X[t2];
Mem[address, dbytes, AccType_NORMAL] = data1;
Mem[address+dbytes, dbytes, AccType_NORMAL] = data2;

if wback then
    if postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.225 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* on page C1-143.

Post-index



32-bit variant

Applies when `size == 10`.

```
STR <Wt>, [<Xn|SP>], #<simm>
```

64-bit variant

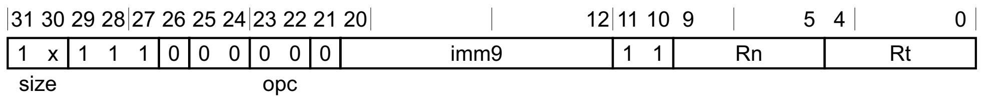
Applies when `size == 11`.

```
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);
```

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

Unsigned offset

31	30	29	28	27	26	25	24	23	22	21 10	9 5	4 0
1	x	1	1	1	0	0	1	0	0	imm12	Rn	Rt
size								opc				

32-bit 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);

integer datasize = 8 << scale;
```

Operation for all encodings

```
bits(64) address;
bits(datasize) data;
boolean rt_unknown = FALSE;

if wback && n == t && n != 31 then
    c = ConstrainUnpredictable();
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE    rt_unknown = FALSE;    // value stored is original value
        when Constraint_UNKNOWN rt_unknown = TRUE;    // value stored is UNKNOWN
        when Constraint_UNDEF   UnallocatedEncoding();
        when Constraint_NOP     EndOfInstruction();

if n == 31 then
```

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

```
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if !postindex then
    address = address + offset;

if rt_unknown then
    data = bits(datasize) UNKNOWN;
else
    data = X[t];
Mem[address, datasize DIV 8, AccType_NORMAL] = data;

if wback then
    if postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```

C6.2.226 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* on page C1-143.

The instruction uses an offset addressing mode, that calculates the address used for the memory access from a base register value and an offset register value. The offset can be optionally shifted and extended.

31	30	29	28	27	26	25	24	23	22	21	20 16	15 13	12	11	10	9 5	4 0
1	x	1	1	1	0	0	0	0	0	1	Rm	option	S	1	0	Rn	Rt
size								opc									

32-bit 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 UnallocatedEncoding();    // sub-word index
ExtendType extend_type = DecodeRegExtend(option);
integer shift = if S == '1' then scale else 0;
```

Assembler symbols

`<Wt>`	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
`<Xt>`	Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
`<Xn|SP>`	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
`<Wm>`	When `option<0>` is set to `0`, is the 32-bit name of the general-purpose index register, encoded in the "Rm" field.
`<Xm>`	When `option<0>` is set to `1`, is the 64-bit name of the general-purpose index register, encoded in the "Rm" field.
`<extend>`	Is the index extend/shift specifier, defaulting to LSL, and which must be omitted for the LSL option when <amount> is omitted. encoded in 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`

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

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 datasize = 8 << scale;
```

Operation

```
bits(64) offset = ExtendReg(m, extend_type, shift);
bits(64) address;
bits(datasize) data;

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

address = address + offset;

data = X[t];
Mem[address, datasize DIV 8, AccType_NORMAL] = data;
```

C6.2.227 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* on page C1-143.

Post-index

31 30	29 28 27	26	25 24	23 22	21	20 … 12	11 10	9 … 5	4 … 0
0 0	1 1 1	0	0 0	0 0	0	imm9	0 1	Rn	Rt
size				opc					

Post-index variant

```
STRB <Wt>, [<Xn|SP>], #<simm>
```

Decode for this encoding

```
boolean wback = TRUE;
boolean postindex = TRUE;
bits(64) offset = SignExtend(imm9, 64);
```

Pre-index

31 30	29 28 27	26	25 24	23 22	21	20 … 12	11 10	9 … 5	4 … 0
0 0	1 1 1	0	0 0	0 0	0	imm9	1 1	Rn	Rt
size				opc					

Pre-index variant

```
STRB <Wt>, [<Xn|SP>, #<simm>]!
```

Decode for this encoding

```
boolean wback = TRUE;
boolean postindex = FALSE;
bits(64) offset = SignExtend(imm9, 64);
```

Unsigned offset

31 30	29 28 27	26	25 24	23 22	21 … 10	9 … 5	4 … 0
0 0	1 1 1	0	0 1	0 0	imm12	Rn	Rt
size				opc			

Unsigned offset variant

```
STRB <Wt>, [<Xn|SP>{, #<pimm>}]
```

Decode for this encoding

```
boolean wback = FALSE;
boolean postindex = FALSE;
bits(64) offset = LSL(ZeroExtend(imm12, 64), 0);
```

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

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)* on page K1-6131.

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);
```

Operation for all encodings

```
bits(64) address;
bits(8) data;
boolean rt_unknown = FALSE;

if wback && n == t && n != 31 then
    c = ConstrainUnpredictable();
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE    rt_unknown = FALSE;    // value stored is original value
        when Constraint_UNKNOWN rt_unknown = TRUE;    // value stored is UNKNOWN
        when Constraint_UNDEF   UnallocatedEncoding();
        when Constraint_NOP     EndOfInstruction();

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

if !postindex then
    address = address + offset;

if rt_unknown then
    data = bits(8) UNKNOWN;
else
    data = X[t];
Mem[address, 1, AccType_NORMAL] = data;

if wback then
    if postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.228 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* on page C1-143.

The instruction uses an offset addressing mode, that calculates the address used for the memory access from a base register value and an offset register value. The offset can be optionally shifted and extended.

31 30	29 28 27	26	25 24	23 22	21	20 16	15 13	12	11 10	9 5	4 0
0 0	1 1 1	0	0 0	0 0	1	Rm	option	S	1 0	Rn	Rt
size				opc							

Extended register 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 UnallocatedEncoding();    // 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);
```

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
bits(64) offset = ExtendReg(m, extend_type, 0);
bits(64) address;
bits(8) data;

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

address = address + offset;

data = X[t];
Mem[address, 1, AccType_NORMAL] = data;
```

C6.2.229 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* on page C1-143.

Post-index

31	30	29	28	27	26	25	24	23	22	21	20 … 12	11	10	9 … 5	4 … 0
0	1	1	1	1	0	0	0	0	0	0	imm9	0	1	Rn	Rt
size								opc							

Post-index variant

```
STRH <Wt>, [<Xn|SP>], #<simm>
```

Decode for this encoding

```
boolean wback = TRUE;
boolean postindex = TRUE;
bits(64) offset = SignExtend(imm9, 64);
```

Pre-index

31	30	29	28	27	26	25	24	23	22	21	20 … 12	11	10	9 … 5	4 … 0
0	1	1	1	1	0	0	0	0	0	0	imm9	1	1	Rn	Rt
size								opc							

Pre-index variant

```
STRH <Wt>, [<Xn|SP>, #<simm>]!
```

Decode for this encoding

```
boolean wback = TRUE;
boolean postindex = FALSE;
bits(64) offset = SignExtend(imm9, 64);
```

Unsigned offset

31	30	29	28	27	26	25	24	23	22	21 … 10	9 … 5	4 … 0
0	1	1	1	1	0	0	1	0	0	imm12	Rn	Rt
size								opc				

Unsigned offset variant

```
STRH <Wt>, [<Xn|SP>{, #<pimm>}]
```

Decode for this encoding

```
boolean wback = FALSE;
boolean postindex = FALSE;
bits(64) offset = LSL(ZeroExtend(imm12, 64), 1);
```

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

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)* on page K1-6131.

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);
```

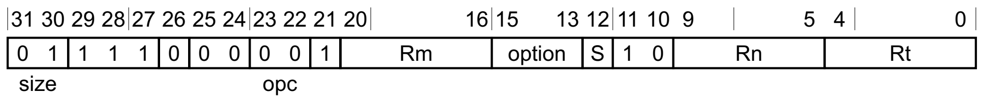
Operation for all encodings

```
bits(64) address;
bits(16) data;
boolean rt_unknown = FALSE;

if wback && n == t && n != 31 then
    c = ConstrainUnpredictable();
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE    rt_unknown = FALSE;    // value stored is original value
        when Constraint_UNKNOWN rt_unknown = TRUE;    // value stored is UNKNOWN
        when Constraint_UNDEF   UnallocatedEncoding();
        when Constraint_NOP     EndOfInstruction();

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

if !postindex then
    address = address + offset;

if rt_unknown then
    data = bits(16) UNKNOWN;
else
    data = X[t];
Mem[address, 2, AccType_NORMAL] = data;

if wback then
    if postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```

C6.2.230 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* on page C1-143.

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

```
STRH <Wt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

Decode for this encoding

```
if option<1> == '0' then UnallocatedEncoding();    // 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);
```

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
bits(64) offset = ExtendReg(m, extend_type, shift);
bits(64) address;
bits(16) data;

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

address = address + offset;

data = X[t];
Mem[address, 2, AccType_NORMAL] = data;
```

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.231 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` has no memory ordering semantics.
- `STSETLB` stores to memory with release semantics, as described in *Load-Acquire, Store-Release* on page B2-94.

For information about memory accesses see *Load/Store addressing modes* on page C1-143.

ARMv8.1

31 30	29 28 27	26	25 24	23	22	21	20 … 16	15	14 … 12	11 10	9 … 5	4 3 2 1 0
0 0	1 1 1	0	0 0	0	R	1	Rs	0	0 1 1	0 0	Rn	1 1 1 1 1
size		V		A				o3	opc			

No memory ordering variant

Applies when `R == 0`.

```
STSETB <Ws>, [<Xn|SP>]
```

Release variant

Applies when `R == 1`.

```
STSETLB <Ws>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !HaveAtomicExt() then UnallocatedEncoding();
integer n = UInt(Rn);
integer s = UInt(Rs);

AccType stacctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
```

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

```
bits(64) address;
bits(8) value;
bits(8) data;
bits(8) result;

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, 1, AccType_ATOMICRW];

result = data OR value;
```

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

```
        // All observers in the shareability domain observe the
        // following load and store atomically.
        Mem[address, 1, stacctype] = result;
```

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.232 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` has no memory ordering semantics.
- `STSETLH` stores to memory with release semantics, as described in *Load-Acquire, Store-Release* on page B2-94.

For information about memory accesses see *Load/Store addressing modes* on page C1-143.

ARMv8.1

31 30	29 28 27	26	25 24	23	22	21	20 16	15	14 12	11 10	9 5	4 3 2 1 0
0 1	1 1 1	0	0 0	0	R	1	Rs	0	0 1 1	0 0	Rn	1 1 1 1 1
size		V		A				o3	opc			

No memory ordering variant

Applies when `R == 0`.

`STSETH <Ws>, [<Xn|SP>]`

Release variant

Applies when `R == 1`.

`STSETLH <Ws>, [<Xn|SP>]`

Decode for all variants of this encoding

```
if !HaveAtomicExt() then UnallocatedEncoding();
integer n = UInt(Rn);
integer s = UInt(Rs);

AccType stacctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
```

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

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

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, 2, AccType_ATOMICRW];

result = data OR value;
```

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

```
// All observers in the shareability domain observe the
// following load and store atomically.
Mem[address, 2, stacctype] = result;
```

C6.2.233 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` has no memory ordering semantics.
- `STSETL` stores to memory with release semantics, as described in *Load-Acquire, Store-Release* on page B2-94.

For information about memory accesses see *Load/Store addressing modes* on page C1-143.

ARMv8.1

31 30	29 28 27	26	25 24	23	22	21	20 16	15	14 12	11 10	9 5	4 3 2 1 0
1 x	1 1 1	0	0 0	0	R	1	Rs	0	0 1 1	0 0	Rn	1 1 1 1 1
size		V		A				o3	opc			

32-bit, no memory ordering variant

Applies when `size == 10 && R == 0`.

```
STSET <Ws>, [<Xn|SP>]
```

32-bit, release variant

Applies when `size == 10 && R == 1`.

```
STSETL <Ws>, [<Xn|SP>]
```

64-bit, no memory ordering variant

Applies when `size == 11 && R == 0`.

```
STSET <Xs>, [<Xn|SP>]
```

64-bit, release variant

Applies when `size == 11 && R == 1`.

```
STSETL <Xs>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !HaveAtomicExt() then UnallocatedEncoding();
integer n = UInt(Rn);
integer s = UInt(Rs);

integer datasize = 8 << UInt(size);
AccType stacctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
```

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.

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

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

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, datasize DIV 8, AccType_ATOMICRW];

result = data OR value;
// All observers in the shareability domain observe the
// following load and store atomically.
Mem[address, datasize DIV 8, stacctype] = result;
```

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.234 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` has no memory ordering semantics.
- `STSMAXLB` stores to memory with release semantics, as described in *Load-Acquire, Store-Release* on page B2-94.

For information about memory accesses see *Load/Store addressing modes* on page C1-143.

ARMv8.1

31 30	29 28 27	26	25 24	23	22	21	20 … 16	15	14 … 12	11 10	9 … 5	4 3 2 1 0
0 0	1 1 1	0	0 0	0	R	1	Rs	0	1 0 0	0 0	Rn	1 1 1 1 1
size		V		A				o3	opc			

No memory ordering variant

Applies when `R == 0`.

```
STSMAXB <Ws>, [<Xn|SP>]
```

Release variant

Applies when `R == 1`.

```
STSMAXLB <Ws>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !HaveAtomicExt() then UnallocatedEncoding();
integer n = UInt(Rn);
integer s = UInt(Rs);

AccType stacctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
```

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

```
bits(64) address;
bits(8) value;
bits(8) data;
bits(8) result;

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, 1, AccType_ATOMICRW];
```

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

```
result = if SInt(data) > SInt(value) then data else value;
// All observers in the shareability domain observe the
// following load and store atomically.
Mem[address, 1, stacctype] = result;
```

C6.2.235 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` has no memory ordering semantics.
- `STSMAXLH` stores to memory with release semantics, as described in *Load-Acquire, Store-Release* on page B2-94.

For information about memory accesses see *Load/Store addressing modes* on page C1-143.

ARMv8.1

31 30	29 28 27	26	25 24	23	22	21	20 … 16	15	14 … 12	11 10	9 … 5	4 3 2 1 0
0 1	1 1 1	0	0 0	0	R	1	Rs	0	1 0 0	0 0	Rn	1 1 1 1 1
size		V		A				o3	opc			

No memory ordering variant

Applies when `R == 0`.

```
STSMAXH <Ws>, [<Xn|SP>]
```

Release variant

Applies when `R == 1`.

```
STSMAXLH <Ws>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !HaveAtomicExt() then UnallocatedEncoding();
integer n = UInt(Rn);
integer s = UInt(Rs);

AccType stacctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
```

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

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

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, 2, AccType_ATOMICRW];
```

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

```
result = if SInt(data) > SInt(value) then data else value;
// All observers in the shareability domain observe the
// following load and store atomically.
Mem[address, 2, stacctype] = result;
```

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.236 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` has no memory ordering semantics.
- `STSMAXL` stores to memory with release semantics, as described in *Load-Acquire, Store-Release* on page B2-94.

For information about memory accesses see *Load/Store addressing modes* on page C1-143.

ARMv8.1

31 30	29 28 27	26	25 24	23	22	21	20 16	15	14 12	11 10	9 5	4 3 2 1 0
1 x	1 1 1	0	0 0	0	R	1	Rs	0	1 0 0	0 0	Rn	1 1 1 1 1
size		V		A				o3	opc			

32-bit, no memory ordering variant

Applies when `size == 10 && R == 0`.

`STSMAX <Ws>, [<Xn|SP>]`

32-bit, release variant

Applies when `size == 10 && R == 1`.

`STSMAXL <Ws>, [<Xn|SP>]`

64-bit, no memory ordering variant

Applies when `size == 11 && R == 0`.

`STSMAX <Xs>, [<Xn|SP>]`

64-bit, release variant

Applies when `size == 11 && R == 1`.

`STSMAXL <Xs>, [<Xn|SP>]`

Decode for all variants of this encoding

```
if !HaveAtomicExt() then UnallocatedEncoding();
integer n = UInt(Rn);
integer s = UInt(Rs);

integer datasize = 8 << UInt(size);
AccType stacctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
```

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.

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

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

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, datasize DIV 8, AccType_ATOMICRW];

result = if SInt(data) > SInt(value) then data else value;
// All observers in the shareability domain observe the
// following load and store atomically.
Mem[address, datasize DIV 8, stacctype] = result;
```

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.237 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` has no memory ordering semantics.
- `STSMINLB` stores to memory with release semantics, as described in *Load-Acquire, Store-Release* on page B2-94.

For information about memory accesses see *Load/Store addressing modes* on page C1-143.

ARMv8.1

31 30	29 28 27	26	25 24	23	22	21	20 16	15	14 12	11 10	9 5	4 3 2 1 0
0 0	1 1 1	0	0 0	0	R	1	Rs	0	1 0 0	0 0	Rn	1 1 1 1 1
size		V		A				o3	opc			

No memory ordering variant

Applies when `R == 0`.

```
STSMINB <Ws>, [<Xn|SP>]
```

Release variant

Applies when `R == 1`.

```
STSMINLB <Ws>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !HaveAtomicExt() then UnallocatedEncoding();
integer n = UInt(Rn);
integer s = UInt(Rs);

AccType stacctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
```

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

```
bits(64) address;
bits(8) value;
bits(8) data;
bits(8) result;

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, 1, AccType_ATOMICRW];
```

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

```
result = if SInt(data) > SInt(value) then value else data;
// All observers in the shareability domain observe the
// following load and store atomically.
Mem[address, 1, stacctype] = result;
```

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.238 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` has no memory ordering semantics.
- `STSMINLH` stores to memory with release semantics, as described in *Load-Acquire, Store-Release* on page B2-94.

For information about memory accesses see *Load/Store addressing modes* on page C1-143.

ARMv8.1

31 30	29 28 27	26	25 24	23	22	21	20 16	15	14 12	11 10	9 5	4 3 2 1 0
0 1	1 1 1	0	0 0	0	R	1	Rs	0	1 0 1	0 0	Rn	1 1 1 1 1
size		V		A				o3	opc			

No memory ordering variant

Applies when `R == 0`.

```
STSMINH <Ws>, [<Xn|SP>]
```

Release variant

Applies when `R == 1`.

```
STSMINLH <Ws>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !HaveAtomicExt() then UnallocatedEncoding();
integer n = UInt(Rn);
integer s = UInt(Rs);

AccType stacctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
```

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

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

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, 2, AccType_ATOMICRW];
```

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

```
result = if SInt(data) > SInt(value) then value else data;
// All observers in the shareability domain observe the
// following load and store atomically.
Mem[address, 2, stacctype] = result;
```

C6.2.239 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` has no memory ordering semantics.
- `STSMINL` stores to memory with release semantics, as described in *Load-Acquire, Store-Release* on page B2-94.

For information about memory accesses see *Load/Store addressing modes* on page C1-143.

ARMv8.1

31 30	29 28 27	26	25 24	23	22	21	20 16	15	14 12	11 10	9 5	4 3 2 1 0
1 x	1 1 1	0	0 0	0	R	1	Rs	0	1 0 1	0 0	Rn	1 1 1 1 1
size		V		A				o3	opc			

32-bit, no memory ordering variant

Applies when `size == 10 && R == 0`.

```
STSMIN <Ws>, [<Xn|SP>]
```

32-bit, release variant

Applies when `size == 10 && R == 1`.

```
STSMINL <Ws>, [<Xn|SP>]
```

64-bit, no memory ordering variant

Applies when `size == 11 && R == 0`.

```
STSMIN <Xs>, [<Xn|SP>]
```

64-bit, release variant

Applies when `size == 11 && R == 1`.

```
STSMINL <Xs>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !HaveAtomicExt() then UnallocatedEncoding();
integer n = UInt(Rn);
integer s = UInt(Rs);

integer datasize = 8 << UInt(size);
AccType stacctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
```

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.

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

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

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, datasize DIV 8, AccType_ATOMICRW];

result = if SInt(data) > SInt(value) then value else data;
// All observers in the shareability domain observe the
// following load and store atomically.
Mem[address, datasize DIV 8, stacctype] = result;
```

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.240 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.

The memory is restricted as if execution is at EL0 when:

- Executing at EL1.
- Executing at EL2, in ARMv8.1, with HCR_EL2.{E2H, TGE} set to {1, 1}.

Otherwise, the access permission is for the Exception level at which the instruction is executed. For information about memory accesses, see *Load/Store addressing modes* on page C1-143.

31 30	29 28 27	26	25 24	23 22	21	20 … 12	11 10	9 … 5	4 … 0
1 x	1 1 1	0	0 0	0 0	0	imm9	1 0	Rn	Rt
size				opc					

32-bit 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);

integer datasize = 8 << scale;
```

Operation

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

if n == 31 then
    CheckSPAlignment();
```

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

```
    address = SP[];
else
    address = X[n];

address = address + offset;

data = X[t];
Mem[address, datasize DIV 8, AccType_UNPRIV] = data;
```

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.241 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.

The memory is restricted as if execution is at EL0 when:

- Executing at EL1.
- Executing at EL2, in ARMv8.1, with HCR_EL2.{E2H, TGE} set to {1, 1}.

Otherwise, the access permission is for the Exception level at which the instruction is executed. For information about memory accesses, see *Load/Store addressing modes* on page C1-143.

31 30	29 28 27	26	25 24	23 22	21	20 … 12	11 10	9 … 5	4 … 0
0 0	1 1 1	0	0 0	0 0	0	imm9	1 0	Rn	Rt
size				opc					

Unscaled offset variant

```
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);
```

Operation

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

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

address = address + offset;

data = X[t];
Mem[address, 1, AccType_UNPRIV] = data;
```

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

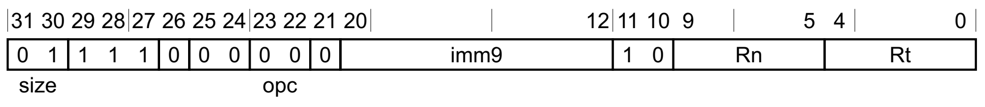
C6.2.242 STTRH

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.

The memory is restricted as if execution is at EL0 when:

- Executing at EL1.
- Executing at EL2, in ARMv8.1, with HCR_EL2.{E2H, TGE} set to {1, 1}.

Otherwise, the access permission is for the Exception level at which the instruction is executed. For information about memory accesses, see *Load/Store addressing modes* on page C1-143.



Unscaled offset variant

```
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);
```

Operation

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

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

address = address + offset;

data = X[t];
Mem[address, 2, AccType_UNPRIV] = data;
```

C6.2.243 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` has no memory ordering semantics.
- `STUMAXLB` stores to memory with release semantics, as described in *Load-Acquire, Store-Release* on page B2-94.

For information about memory accesses see *Load/Store addressing modes* on page C1-143.

ARMv8.1

31 30	29 28 27	26	25 24	23	22	21	20 16	15	14 12	11 10	9 5	4 3 2 1 0
0 0	1 1 1	0	0 0	0	R	1	Rs	0	1 1 0	0 0	Rn	1 1 1 1 1
size		V		A				o3	opc			

No memory ordering variant

Applies when `R == 0`.

`STUMAXB <Ws>, [<Xn|SP>]`

Release variant

Applies when `R == 1`.

`STUMAXLB <Ws>, [<Xn|SP>]`

Decode for all variants of this encoding

```
if !HaveAtomicExt() then UnallocatedEncoding();
integer n = UInt(Rn);
integer s = UInt(Rs);

AccType stacctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
```

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

```
bits(64) address;
bits(8) value;
bits(8) data;
bits(8) result;

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, 1, AccType_ATOMICRW];
```

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

```
result = if UInt(data) > UInt(value) then data else value;
// All observers in the shareability domain observe the
// following load and store atomically.
Mem[address, 1, stacctype] = result;
```

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.244 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` has no memory ordering semantics.
- `STUMAXLH` stores to memory with release semantics, as described in *Load-Acquire, Store-Release* on page B2-94.

For information about memory accesses see *Load/Store addressing modes* on page C1-143.

ARMv8.1

31 30	29 28 27	26	25 24	23	22	21	20 16	15	14 12	11 10	9 5	4 3 2 1 0
0 1	1 1 1	0	0 0	0	R	1	Rs	0	1 1 0	0 0	Rn	1 1 1 1 1
size		V		A				o3	opc			

No memory ordering variant

Applies when `R == 0`.

```
STUMAXH <Ws>, [<Xn|SP>]
```

Release variant

Applies when `R == 1`.

```
STUMAXLH <Ws>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !HaveAtomicExt() then UnallocatedEncoding();
integer n = UInt(Rn);
integer s = UInt(Rs);

AccType stacctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
```

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

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

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, 2, AccType_ATOMICRW];
```

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

```
result = if UInt(data) > UInt(value) then data else value;
// All observers in the shareability domain observe the
// following load and store atomically.
Mem[address, 2, stacctype] = result;
```

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.245 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` has no memory ordering semantics.
- `STUMAXL` stores to memory with release semantics, as described in *Load-Acquire, Store-Release* on page B2-94.

For information about memory accesses see *Load/Store addressing modes* on page C1-143.

ARMv8.1

31	30	29	28	27	26	25	24	23	22	21	20–16	15	14	13	12	11	10	9–5	4	3	2	1	0
1	x	1	1	1	0	0	0	0	R	1	Rs	0	1	1	0	0	0	Rn	1	1	1	1	1
size					V			A				o3	opc										

32-bit, no memory ordering variant

Applies when `size == 10 && R == 0`.

```
STUMAX <Ws>, [<Xn|SP>]
```

32-bit, release variant

Applies when `size == 10 && R == 1`.

```
STUMAXL <Ws>, [<Xn|SP>]
```

64-bit, no memory ordering variant

Applies when `size == 11 && R == 0`.

```
STUMAX <Xs>, [<Xn|SP>]
```

64-bit, release variant

Applies when `size == 11 && R == 1`.

```
STUMAXL <Xs>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !HaveAtomicExt() then UnallocatedEncoding();
integer n = UInt(Rn);
integer s = UInt(Rs);

integer datasize = 8 << UInt(size);
AccType stacctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
```

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.

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

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

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, datasize DIV 8, AccType_ATOMICRW];

result = if UInt(data) > UInt(value) then data else value;
// All observers in the shareability domain observe the
// following load and store atomically.
Mem[address, datasize DIV 8, stacctype] = result;
```

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.246 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 has no memory ordering semantics.
- STUMINLB stores to memory with release semantics, as described in *Load-Acquire, Store-Release* on page B2-94.

For information about memory accesses see *Load/Store addressing modes* on page C1-143.

ARMv8.1

31 30	29 28 27	26	25 24	23	22	21	20 16	15	14 12	11 10	9 5	4 3 2 1 0
0 0	1 1 1	0	0 0	0	R	1	Rs	0	1 1 1	0 0	Rn	1 1 1 1 1
size		V		A				o3	opc			

No memory ordering variant

Applies when R == 0.

```
STUMINB <Ws>, [<Xn|SP>]
```

Release variant

Applies when R == 1.

```
STUMINLB <Ws>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !HaveAtomicExt() then UnallocatedEncoding();
integer n = UInt(Rn);
integer s = UInt(Rs);

AccType stacctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
```

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

```
bits(64) address;
bits(8) value;
bits(8) data;
bits(8) result;

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, 1, AccType_ATOMICRW];
```

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

```
result = if UInt(data) > UInt(value) then value else data;
// All observers in the shareability domain observe the
// following load and store atomically.
Mem[address, 1, stacctype] = result;
```

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.247 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` has no memory ordering semantics.
- `STUMINLH` stores to memory with release semantics, as described in *Load-Acquire, Store-Release* on page B2-94.

For information about memory accesses see *Load/Store addressing modes* on page C1-143.

ARMv8.1

31 30	29 28 27	26	25 24	23	22	21	20 16	15	14 12	11 10	9 5	4 3 2 1 0
0 1	1 1 1	0	0 0	0	R	1	Rs	0	1 1 1	0 0	Rn	1 1 1 1 1
size		V		A				o3	opc			

No memory ordering variant

Applies when `R == 0`.

`STUMINH <Ws>, [<Xn|SP>]`

Release variant

Applies when `R == 1`.

`STUMINLH <Ws>, [<Xn|SP>]`

Decode for all variants of this encoding

```
if !HaveAtomicExt() then UnallocatedEncoding();
integer n = UInt(Rn);
integer s = UInt(Rs);

AccType stacctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
```

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

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

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, 2, AccType_ATOMICRW];
```

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

```
result = if UInt(data) > UInt(value) then value else data;
// All observers in the shareability domain observe the
// following load and store atomically.
Mem[address, 2, stacctype] = result;
```

C6.2.248 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` has no memory ordering semantics.
- `STUMINL` stores to memory with release semantics, as described in *Load-Acquire, Store-Release* on page B2-94.

For information about memory accesses see *Load/Store addressing modes* on page C1-143.

ARMv8.1

31 30	29 28 27	26	25 24	23	22	21	20 16	15	14 12	11 10	9 5	4 3 2 1 0
1 x	1 1 1	0	0 0	0	R	1	Rs	0	1 1 1	0 0	Rn	1 1 1 1 1
size		V		A				o3	opc			

32-bit, no memory ordering variant

Applies when `size == 10 && R == 0`.

```
STUMIN <Ws>, [<Xn|SP>]
```

32-bit, release variant

Applies when `size == 10 && R == 1`.

```
STUMINL <Ws>, [<Xn|SP>]
```

64-bit, no memory ordering variant

Applies when `size == 11 && R == 0`.

```
STUMIN <Xs>, [<Xn|SP>]
```

64-bit, release variant

Applies when `size == 11 && R == 1`.

```
STUMINL <Xs>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !HaveAtomicExt() then UnallocatedEncoding();
integer n = UInt(Rn);
integer s = UInt(Rs);

integer datasize = 8 << UInt(size);
AccType stacctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
```

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.

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

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

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, datasize DIV 8, AccType_ATOMICRW];

result = if UInt(data) > UInt(value) then value else data;
// All observers in the shareability domain observe the
// following load and store atomically.
Mem[address, datasize DIV 8, stacctype] = result;
```

C6.2.249 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* on page C1-143.

31 30	29 28 27	26	25 24	23 22	21	20 … 12	11 10	9 … 5	4 … 0
1 x	1 1 1	0	0 0	0 0	0	imm9	0 0	Rn	Rt
size				opc					

32-bit 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);

integer datasize = 8 << scale;
```

Operation

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

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

address = address + offset;

data = X[t];
Mem[address, datasize DIV 8, AccType_NORMAL] = data;
```

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.250 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* on page C1-143.

31 30	29 28 27	26	25 24	23 22	21	20 12	11 10	9 5	4 0
0 0	1 1 1	0	0 0	0 0	0	imm9	0 0	Rn	Rt
size				opc					

Unscaled offset variant

```
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);
```

Operation

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

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

address = address + offset;

data = X[t];
Mem[address, 1, AccType_NORMAL] = data;
```

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.251 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* on page C1-143.

31 30	29 28 27	26	25 24	23 22	21	20 … 12	11 10	9 … 5	4 … 0
0 1	1 1 1	0	0 0	0 0	0	imm9	0 0	Rn	Rt
size				opc					

Unscaled offset variant

```
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);
```

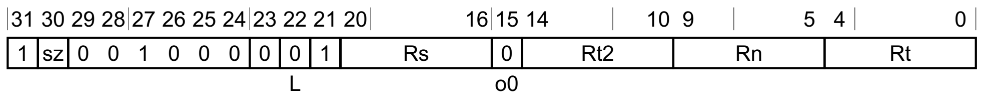
Operation

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

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

address = address + offset;

data = X[t];
Mem[address, 2, AccType_NORMAL] = data;
```

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.252 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* on page B2-121. A 32-bit pair requires the address to be doubleword aligned and is single-copy atomic at doubleword granularity. A 64-bit pair requires the address to be quadword aligned and, if the Store-Exclusive succeeds, it causes a single-copy atomic update of the 128-bit memory location being updated. For information about memory accesses see *Load/Store addressing modes* on page C1-143.



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

integer elsize = 32 << UInt(sz);
integer datasize = elsize * 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 *STXP* on page K1-6131.

Assembler symbols

`<Ws>`	Is the 32-bit name of the general-purpose register into which the status result of the store exclusive is written, encoded in the "Rs" field. The value returned is: `0` If the operation updates memory. `1` If the operation fails to update memory.
`<Xt1>`	Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.
`<Xt2>`	Is the 64-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.
`<Wt1>`	Is the 32-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.
`<Wt2>`	Is the 32-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.
`<Xn|SP>`	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Aborts and alignment

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- `<Ws>` is not updated.

Accessing an address that is not aligned to the size of the data being accessed causes an Alignment fault Data Abort exception to be generated, subject to the following rules:

- If `AArch64.ExclusiveMonitorsPass()` returns `TRUE`, the exception is generated.
- Otherwise, it is IMPLEMENTATION DEFINED whether the exception is generated.

If `AArch64.ExclusiveMonitorsPass()` returns `FALSE` and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.

Operation

```
bits(64) address;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;

if s == t || (s == t2) then
    Constraint c = ConstrainUnpredictable();
    assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN rt_unknown = TRUE;    // store UNKNOWN value
        when Constraint_NONE    rt_unknown = FALSE;    // store original value
        when Constraint_UNDEF   UnallocatedEncoding();
        when Constraint_NOP     EndOfInstruction();
if s == n && n != 31 then
    Constraint c = ConstrainUnpredictable();
    assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN rn_unknown = TRUE;    // address is UNKNOWN
        when Constraint_NONE    rn_unknown = FALSE;    // address is original base
        when Constraint_UNDEF   UnallocatedEncoding();
        when Constraint_NOP     EndOfInstruction();

if n == 31 then
    CheckSPAlignment();
    address = SP[];
elsif rn_unknown then
    address = bits(64) UNKNOWN;
else
    address = X[n];

if rt_unknown then
    data = bits(datasize) UNKNOWN;
else
    bits(datasize DIV 2) el1 = X[t];
    bits(datasize DIV 2) el2 = X[t2];
    data = if BigEndian() then el1:el2 else el2:el1;
bit status = '1';
// Check whether the Exclusive Monitors are set to include the
// physical memory locations corresponding to virtual address
// range [address, address+dbytes-1].
if AArch64.ExclusiveMonitorsPass(address, dbytes) then
    // This atomic write will be rejected if it does not refer
    // to the same physical locations after address translation.
    Mem[address, dbytes, AccType_ATOMIC] = data;
    status = ExclusiveMonitorsStatus();
X[s] = ZeroExtend(status, 32);
```

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.253 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* on page B2-121. For information about memory accesses see *Load/Store addressing modes* on page C1-143.

31	30	29	28	27	26	25	24	23	22	21	20 … 16	15	14 … 10	9 … 5	4 … 0
1	x	0	0	1	0	0	0	0	0	0	Rs	0	(1)(1)(1)(1)(1)	Rn	Rt
size									L			o0	Rt2		

32-bit 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

integer elsize = 8 << UInt(size);
```

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* on page K1-6132.

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.

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

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;
boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;

if s == t then
    Constraint c = ConstrainUnpredictable();
    assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN rt_unknown = TRUE;    // store UNKNOWN value
        when Constraint_NONE    rt_unknown = FALSE;    // store original value
        when Constraint_UNDEF   UnallocatedEncoding();
        when Constraint_NOP     EndOfInstruction();
if s == n && n != 31 then
    Constraint c = ConstrainUnpredictable();
    assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN rn_unknown = TRUE;    // address is UNKNOWN
        when Constraint_NONE    rn_unknown = FALSE;    // address is original base
        when Constraint_UNDEF   UnallocatedEncoding();
        when Constraint_NOP     EndOfInstruction();

if n == 31 then
    CheckSPAlignment();
    address = SP[];
elsif rn_unknown then
    address = bits(64) UNKNOWN;
else
    address = X[n];

if rt_unknown then
    data = bits(elsize) UNKNOWN;
else
    data = X[t];

bit status = '1';
// Check whether the Exclusive Monitors are set to include the
// physical memory locations corresponding to virtual address
// range [address, address+dbytes-1].
if AArch64.ExclusiveMonitorsPass(address, dbytes) then
    // This atomic write will be rejected if it does not refer
    // to the same physical locations after address translation.
    Mem[address, dbytes, AccType_ATOMIC] = data;
    status = ExclusiveMonitorsStatus();
X[s] = ZeroExtend(status, 32);
```

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.254 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* on page B2-121. The memory access is atomic.

For information about memory accesses see *Load/Store addressing modes* on page C1-143.

31	30	29	28	27	26	25	24	23	22	21	20 16	15	14 10	9 5	4 0
0	0	0	0	1	0	0	0	0	0	0	Rs	0	(1) (1) (1) (1) (1)	Rn	Rt
size									L			o0	Rt2		

No offset variant

```
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
```

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* on page K1-6132.

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;
boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;

if s == t then
    Constraint c = ConstrainUnpredictable();
    assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
```

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

```
    case c of
        when Constraint_UNKNOWN rt_unknown = TRUE;    // store UNKNOWN value
        when Constraint_NONE    rt_unknown = FALSE;    // store original value
        when Constraint_UNDEF   UnallocatedEncoding();
        when Constraint_NOP     EndOfInstruction();
if s == n && n != 31 then
    Constraint c = ConstrainUnpredictable();
    assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN rn_unknown = TRUE;    // address is UNKNOWN
        when Constraint_NONE    rn_unknown = FALSE;    // address is original base
        when Constraint_UNDEF   UnallocatedEncoding();
        when Constraint_NOP     EndOfInstruction();

if n == 31 then
    CheckSPAlignment();
    address = SP[];
elsif rn_unknown then
    address = bits(64) UNKNOWN;
else
    address = X[n];

if rt_unknown then
    data = bits(8) UNKNOWN;
else
    data = X[t];

bit status = '1';
// Check whether the Exclusive Monitors are set to include the
// physical memory locations corresponding to virtual address
// range [address, address+dbytes-1].
if AArch64.ExclusiveMonitorsPass(address, 1) then
    // This atomic write will be rejected if it does not refer
    // to the same physical locations after address translation.
    Mem[address, 1, AccType_ATOMIC] = data;
    status = ExclusiveMonitorsStatus();
X[s] = ZeroExtend(status, 32);
```

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.255 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* on page B2-121. The memory access is atomic.

For information about memory accesses see *Load/Store addressing modes* on page C1-143.

31 30	29 28 27 26 25 24	23	22	21	20 16	15	14 10	9 5	4 0
0 1	0 0 1 0 0 0	0	0	0	Rs	0	(1) (1) (1) (1) (1)	Rn	Rt
size			L			o0	Rt2		

No offset variant

```
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
```

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;
boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;

if s == t then
    Constraint c = ConstrainUnpredictable();
```

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

```
    assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN rt_unknown = TRUE;    // store UNKNOWN value
        when Constraint_NONE    rt_unknown = FALSE;    // store original value
        when Constraint_UNDEF   UnallocatedEncoding();
        when Constraint_NOP     EndOfInstruction();
if s == n && n != 31 then
    Constraint c = ConstrainUnpredictable();
    assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN rn_unknown = TRUE;    // address is UNKNOWN
        when Constraint_NONE    rn_unknown = FALSE;    // address is original base
        when Constraint_UNDEF   UnallocatedEncoding();
        when Constraint_NOP     EndOfInstruction();

if n == 31 then
    CheckSPAlignment();
    address = SP[];
elsif rn_unknown then
    address = bits(64) UNKNOWN;
else
    address = X[n];

if rt_unknown then
    data = bits(16) UNKNOWN;
else
    data = X[t];

bit status = '1';
// Check whether the Exclusive Monitors are set to include the
// physical memory locations corresponding to virtual address
// range [address, address+dbytes-1].
if AArch64.ExclusiveMonitorsPass(address, 2) then
    // This atomic write will be rejected if it does not refer
    // to the same physical locations after address translation.
    Mem[address, 2, AccType_ATOMIC] = data;
    status = ExclusiveMonitorsStatus();
X[s] = ZeroExtend(status, 32);
```

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.256 SUB (extended register)

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

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

32-bit 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);
integer datasize = if sf == '1' then 64 else 32;
ExtendType extend_type = DecodeRegExtend(option);
integer shift = UInt(imm3);
if shift > 4 then ReservedValue();
```

Assembler symbols

`<Wd|WSP>`	Is the 32-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.
`<Wn|WSP>`	Is the 32-bit name of the first source general-purpose register or stack pointer, encoded in the "Rn" field.
`<Wm>`	Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
`<Xd|SP>`	Is the 64-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.
`<Xn|SP>`	Is the 64-bit name of the first source general-purpose register or stack pointer, encoded in the "Rn" field.
`<R>`	Is a width specifier, encoded in 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[] else X[n];
bits(datasize) operand2 = ExtendReg(m, extend_type, shift);

operand2 = NOT(operand2);
(result, -) = AddWithCarry(operand1, operand2, '1');

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

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.257 SUB (immediate)

Subtract (immediate) subtracts an optionally-shifted immediate value from a register value, and writes the result to the destination register.

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

32-bit 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);
integer datasize = if sf == '1' then 64 else 32;
bits(datasize) imm;

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

Assembler symbols

`<Wd|WSP>`	Is the 32-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.
`<Wn|WSP>`	Is the 32-bit name of the source general-purpose register or stack pointer, encoded in the "Rn" field.
`<Xd|SP>`	Is the 64-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.
`<Xn|SP>`	Is the 64-bit name of the source general-purpose register or stack pointer, encoded in the "Rn" field.
`<imm>`	Is an unsigned immediate, in the range 0 to 4095, encoded in the "imm12" field.
`<shift>`	Is the optional left shift to apply to the immediate, defaulting to LSL #0 and encoded in the "shift" field. It can have the following values:

`LSL #0`	when `shift = 00`
`LSL #12`	when `shift = 01`

The encoding `shift = 1x` is reserved.

Operation

```
bits(datasize) result;
bits(datasize) operand1 = if n == 31 then SP[] else X[n];
bits(datasize) operand2;
```

```
operand2 = NOT(imm);
(result, -) = AddWithCarry(operand1, operand2, '1');

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

C6.2.258 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.

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

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);
integer datasize = if sf == '1' then 64 else 32;

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

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.

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

`<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];
bits(datasize) operand2 = ShiftReg(m, shift_type, shift_amount);

operand2 = NOT(operand2);
(result, -) = AddWithCarry(operand1, operand2, '1');

X[d] = result;
```

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.259 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.

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

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);
integer datasize = if sf == '1' then 64 else 32;
ExtendType extend_type = DecodeRegExtend(option);
integer shift = UInt(imm3);
if shift > 4 then ReservedValue();
```

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

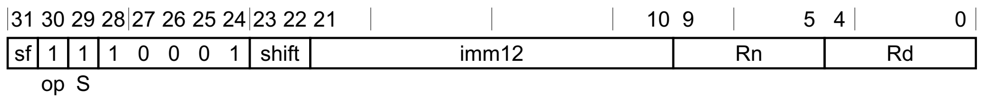
```
bits(datasize) result;
bits(datasize) operand1 = if n == 31 then SP[] else X[n];
bits(datasize) operand2 = ExtendReg(m, extend_type, shift);
bits(4) nzcv;

operand2 = NOT(operand2);
(result, nzcv) = AddWithCarry(operand1, operand2, '1');

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

X[d] = result;
```

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.260 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);
integer datasize = if sf == '1' then 64 else 32;
bits(datasize) imm;

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

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 "shift" field. It can have the following values:

`LSL #0`	when `shift = 00`
`LSL #12`	when `shift = 01`

The encoding `shift = 1x` is reserved.

Operation

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

operand2 = NOT(imm);
(result, nzcv) = AddWithCarry(operand1, operand2, '1');

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

X[d] = result;
```

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.261 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.

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

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);
integer datasize = if sf == '1' then 64 else 32;

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

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

Alias conditions

Alias	is preferred when
CMP (shifted register)	`Rd == '11111'`
NEGS	`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.

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

`<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];
bits(datasize) operand2 = ShiftReg(m, shift_type, shift_amount);
bits(4) nzcv;

operand2 = NOT(operand2);
(result, nzcv) = AddWithCarry(operand1, operand2, '1');

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

X[d] = result;
```

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.262 SVC

Supervisor Call causes an exception to be taken to EL1.

On executing an SVC instruction, the PE records the exception as a Supervisor Call exception in ESR_ELx, using the EC value 0x15, and the value of the immediate argument.

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

System variant

```
SVC #<imm>
```

Decode for this encoding

```
bits(16) imm = imm16;
```

Assembler symbols

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

Operation

```
AArch64.CallSupervisor(imm);
```

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.263 SWPB, SWPAB, SWPALB, SWPLB

Swap byte in memory atomically loads an 8-bit byte from a memory location, and stores the value held in a register back to the same memory location. The value initially loaded from memory is returned in the destination register.

- If the destination register is not `WZR`, `SWPAB` and `SWPALB` load from memory with acquire semantics.
- `SWPLB` and `SWPALB` store to memory with release semantics.
- `SWPB` has no memory ordering requirements.

For more information about memory ordering semantics see *Load-Acquire, Store-Release* on page B2-94.

For information about memory accesses see *Load/Store addressing modes* on page C1-143.

ARMv8.1

31 30	29 28 27	26	25 24	23	22	21	20 … 16	15	14 … 12	11 10	9 … 5	4 … 0
0 0	1 1 1	0	0 0	A	R	1	Rs	1	0 0 0	0 0	Rn	Rt
size		V						o3	opc			

Acquire variant

Applies when `A == 1 && R == 0`.

```
SWPAB <Ws>, <Wt>, [<Xn|SP>]
```

Acquire and release variant

Applies when `A == 1 && R == 1`.

```
SWPALB <Ws>, <Wt>, [<Xn|SP>]
```

No memory ordering variant

Applies when `A == 0 && R == 0`.

```
SWPB <Ws>, <Wt>, [<Xn|SP>]
```

Release variant

Applies when `A == 0 && R == 1`.

```
SWPLB <Ws>, <Wt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !HaveAtomicExt() then UnallocatedEncoding();
integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
```

Assembler symbols

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

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

```
bits(64) address;
bits(8) value;
bits(8) data;

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, 1, ldacctype];

// All observers in the shareability domain observe the
// following load and store atomically.
Mem[address, 1, stacctype] = value;

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

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.264 SWPH, SWPAH, SWPALH, SWPLH

Swap halfword in memory atomically loads a 16-bit halfword from a memory location, and stores the value held in a register back to the same memory location. The value initially loaded from memory is returned in the destination register.

- If the destination register is not `WZR`, `SWPAH` and `SWPALH` load from memory with acquire semantics.
- `SWPLH` and `SWPALH` store to memory with release semantics.
- `SWPH` has no memory ordering requirements.

For more information about memory ordering semantics see *Load-Acquire, Store-Release* on page B2-94.

For information about memory accesses see *Load/Store addressing modes* on page C1-143.

ARMv8.1

31 30	29 28 27	26	25 24	23	22	21	20 16	15	14 13 12	11 10	9 5	4 0
0 1	1 1 1	0	0 0	A	R	1	Rs	1	0 0 0	0 0	Rn	Rt
size		V						o3	opc			

Acquire variant

Applies when `A == 1 && R == 0`.

```
SWPAH <Ws>, <Wt>, [<Xn|SP>]
```

Acquire and release variant

Applies when `A == 1 && R == 1`.

```
SWPALH <Ws>, <Wt>, [<Xn|SP>]
```

No memory ordering variant

Applies when `A == 0 && R == 0`.

```
SWPH <Ws>, <Wt>, [<Xn|SP>]
```

Release variant

Applies when `A == 0 && R == 1`.

```
SWPLH <Ws>, <Wt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !HaveAtomicExt() then UnallocatedEncoding();
integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
```

Assembler symbols

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

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

`<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;

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, 2, ldacctype];

// All observers in the shareability domain observe the
// following load and store atomically.
Mem[address, 2, stacctype] = value;

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

C6.2.265 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 no memory ordering requirements.

For more information about memory ordering semantics see *Load-Acquire, Store-Release* on page B2-94.

For information about memory accesses see *Load/Store addressing modes* on page C1-143.

ARMv8.1

31	30	29	28	27	26	25	24	23	22	21	20 16	15	14		12	11	10	9 5	4 0
1	x	1	1	1	0	0	0	A	R	1	Rs	1	0	0	0	0	0	Rn	Rt
size					V							o3	opc						

32-bit, acquire variant

Applies when `size == 10 && A == 1 && R == 0`.

```
SWPA <Ws>, <Wt>, [<Xn|SP>]
```

32-bit, acquire and release variant

Applies when `size == 10 && A == 1 && R == 1`.

```
SWPAL <Ws>, <Wt>, [<Xn|SP>]
```

32-bit, no memory ordering variant

Applies when `size == 10 && A == 0 && R == 0`.

```
SWP <Ws>, <Wt>, [<Xn|SP>]
```

32-bit, release variant

Applies when `size == 10 && A == 0 && R == 1`.

```
SWPL <Ws>, <Wt>, [<Xn|SP>]
```

64-bit, acquire variant

Applies when `size == 11 && A == 1 && R == 0`.

```
SWPA <Xs>, <Xt>, [<Xn|SP>]
```

64-bit, acquire and release variant

Applies when `size == 11 && A == 1 && R == 1`.

```
SWPAL <Xs>, <Xt>, [<Xn|SP>]
```

64-bit, no memory ordering variant

Applies when `size == 11 && A == 0 && R == 0`.

```
SWP <Xs>, <Xt>, [<Xn|SP>]
```

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

64-bit, release variant

Applies when `size == 11 && A == 0 && R == 1`.

```
SWPL <Xs>, <Xt>, [<Xn|SP>]
```

Decode for all variants of this encoding

```
if !HaveAtomicExt() then UnallocatedEncoding();
integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDRW else AccType_ATOMICRW;
```

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) value;
bits(datasize) data;

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, datasize DIV 8, ldacctype];

// All observers in the shareability domain observe the
// following load and store atomically.
Mem[address, datasize DIV 8, stacctype] = value;

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

C6.2.266 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 for this instruction.

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

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.

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.267 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 for this instruction.

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

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.

C6.2.268 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 for this instruction.

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

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

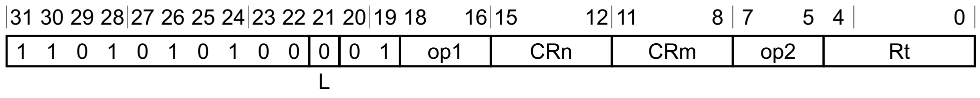
The description of SBFM gives the operational pseudocode for this instruction.

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.269 SYS

System instruction. For more information, see *op0==0b01, cache maintenance, TLB maintenance, and address translation instructions* on page C5-317 for the encodings of System instructions.

This instruction is used by the aliases AT, DC, IC, and TLBI. See *Alias conditions* for details of when each alias is preferred.



System variant

```
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`
DC	`CRn == '0111' && SysOp(op1,'0111',CRm,op2) == Sys_DC`
IC	`CRn == '0111' && SysOp(op1,'0111',CRm,op2) == Sys_IC`
TLBI	`CRn == '1000' && SysOp(op1,'1000',CRm,op2) == Sys_TLBI`

Assembler symbols

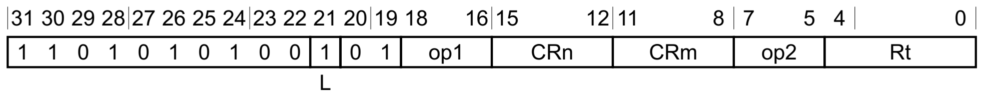
`<op1>`	Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op1" field.
`<Cn>`	Is a name 'Cn', with 'n' in the range 0 to 15, encoded in the "CRn" field.
`<Cm>`	Is a name 'Cm', with 'm' in the range 0 to 15, encoded in the "CRm" field.
`<op2>`	Is a 3-bit unsigned immediate, in the range 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, X[t]);
```

C6.2.270 SYSL

System instruction with result. For more information, see *op0==0b01, cache maintenance, TLB maintenance, and address translation instructions* on page C5-317 for the encodings of System instructions.



System variant

```
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);
```

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.

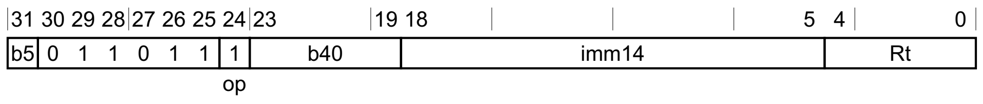
Operation

```
X[t] = AArch64.SysInstrWithResult(1, sys_op1, sys_crn, sys_crm, sys_op2);
```

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.271 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.



14-bit signed PC-relative branch offset variant

```
TBNZ <R><t>, #<imm>, <label>
```

Decode for this encoding

```
integer t = UInt(Rt);

integer datasize = if b5 == '1' then 64 else 32;
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];

if operand<bit_pos> == op then
    BranchTo(PC[] + offset, BranchType_JMP);
```

C6.2.272 TBZ

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

31	30	29	28	27	26	25	24	23 19	18 5	4 0
b5	0	1	1	0	1	1	0	b40	imm14	Rt
							op			

14-bit signed PC-relative branch offset variant

```
TBZ <R><t>, #<imm>, <label>
```

Decode for this encoding

```
integer t = UInt(Rt);

integer datasize = if b5 == '1' then 64 else 32;
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];

if operand<bit_pos> == op then
    BranchTo(PC[] + offset, BranchType_JMP);
```

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.273 TLBI

TLB Invalidate operation. For more information, see A64 system instructions for TLB maintenance.

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 for this instruction.

31 30 29 28 27 26 25 24 23 22	21	20 19	18 16	15 12	11 8	7 5	4 0
1 1 0 1 0 1 0 1 0 0	0	0 1	op1	1 0 0 0	CRm	op2	Rt
	L			CRn			

System variant

`TLBI <tlbi_op>{, <Xt>}`

is equivalent to

`SYS #<op1>, C8, <Cm>, #<op2>{, <Xt>}`

and is the preferred disassembly when `SysOp(op1,'1000',CRm,op2) == Sys_TLBI`.

Assembler symbols

`<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.
`<tlbi_op>`	Is a TLBI instruction name, as listed for the TLBI system instruction group, encoded in the "op1:CRm:op2" field. It can have the following values:

`VMALLE1IS`	when `op1 = 000`, `CRm = 0011`, `op2 = 000`
`VAE1IS`	when `op1 = 000`, `CRm = 0011`, `op2 = 001`
`ASIDE1IS`	when `op1 = 000`, `CRm = 0011`, `op2 = 010`
`VAAE1IS`	when `op1 = 000`, `CRm = 0011`, `op2 = 011`
`VALE1IS`	when `op1 = 000`, `CRm = 0011`, `op2 = 101`
`VAALE1IS`	when `op1 = 000`, `CRm = 0011`, `op2 = 111`
`VMALLE1`	when `op1 = 000`, `CRm = 0111`, `op2 = 000`
`VAE1`	when `op1 = 000`, `CRm = 0111`, `op2 = 001`
`ASIDE1`	when `op1 = 000`, `CRm = 0111`, `op2 = 010`
`VAAE1`	when `op1 = 000`, `CRm = 0111`, `op2 = 011`
`VALE1`	when `op1 = 000`, `CRm = 0111`, `op2 = 101`
`VAALE1`	when `op1 = 000`, `CRm = 0111`, `op2 = 111`
`IPAS2E1IS`	when `op1 = 100`, `CRm = 0000`, `op2 = 001`
`IPAS2LE1IS`	when `op1 = 100`, `CRm = 0000`, `op2 = 101`
`ALLE2IS`	when `op1 = 100`, `CRm = 0011`, `op2 = 000`
`VAE2IS`	when `op1 = 100`, `CRm = 0011`, `op2 = 001`
`ALLE1IS`	when `op1 = 100`, `CRm = 0011`, `op2 = 100`
`VALE2IS`	when `op1 = 100`, `CRm = 0011`, `op2 = 101`

VMALLS12E1IS	when op1 = 100, CRm = 0011, op2 = 110
IPAS2E1	when op1 = 100, CRm = 0100, op2 = 001
IPAS2LE1	when op1 = 100, CRm = 0100, op2 = 101
ALLE2	when op1 = 100, CRm = 0111, op2 = 000
VAE2	when op1 = 100, CRm = 0111, op2 = 001
ALLE1	when op1 = 100, CRm = 0111, op2 = 100
VALE2	when op1 = 100, CRm = 0111, op2 = 101
VMALLS12E1	when op1 = 100, CRm = 0111, op2 = 110
ALLE3IS	when op1 = 110, CRm = 0011, op2 = 000
VAE3IS	when op1 = 110, CRm = 0011, op2 = 001
VALE3IS	when op1 = 110, CRm = 0011, op2 = 101
ALLE3	when op1 = 110, CRm = 0111, op2 = 000
VAE3	when op1 = 110, CRm = 0111, op2 = 001
VALE3	when op1 = 110, CRm = 0111, op2 = 101

<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.

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.274 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 for this instruction.

31	30	29	28	27	26	25	24	23	22	21–16	15–10	9–5	4	3	2	1	0
sf	1	1	1	0	0	1	0	0	N	immr	imms	Rn	1	1	1	1	1
	opc												Rd				

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.

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.275 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 for this instruction.

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

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,

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

The description of ANDS (shifted register) gives the operational pseudocode for this instruction.

C6.2.276 UBFIZ

Unsigned Bitfield Insert in Zero zeroes the destination register and copies any number of contiguous bits from a source register into any position in 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 for this instruction.

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

32-bit 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

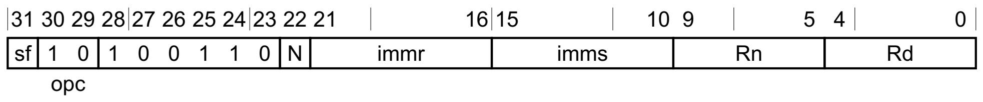
The description of UBFM gives the operational pseudocode for this instruction.

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.277 UBFM

Unsigned Bitfield Move copies any number of low-order bits from a source register into the same number of adjacent bits at any position in the destination register, with zeros in the upper and lower bits.

This instruction is used by the aliases LSL (immediate), LSR (immediate), UBFIZ, UBFX, UXTB, and UXTH. See *Alias conditions* on page C6-961 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);
integer datasize = if sf == '1' then 64 else 32;

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

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

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

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];

// perform bitfield move on low bits
bits(datasize) bot = ROR(src, R) AND wmask;

// combine extension bits and result bits
X[d] = bot AND tmask;
```

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.278 UBFX

Unsigned Bitfield Extract extracts any number of adjacent bits at any position from a register, zero-extends them 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 for this instruction.

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

32-bit 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.

C6.2.279 UDIV

Unsigned Divide divides an unsigned integer register value by another unsigned integer register value, and writes the result to the destination register. The condition flags are not affected.

31	30	29	28	27	26	25	24	23	22	21	20 16	15	14	13	12	11	10	9 5	4 0
sf	0	0	1	1	0	1	0	1	1	0	Rm	0	0	0	0	1	0	Rn	Rd
																	o1		

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);
integer datasize = if sf == '1' then 64 else 32;
```

Assembler symbols

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

Operation

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

if IsZero(operand2) then
    result = 0;
else
    result = RoundTowardsZero(Real(Int(operand1, TRUE)) / Real(Int(operand2, TRUE)));

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

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.280 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.

31	30 29	28 27 26 25 24	23	22 21	20 … 16	15	14 … 10	9 … 5	4 … 0
1	0 0	1 1 0 1 1	1	0 1	Rm	0	Ra	Rn	Rd
			U			o0			

64-bit variant

```
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];
bits(32) operand2 = X[m];
bits(64) operand3 = X[a];

integer result;

result = Int(operand3, TRUE) + (Int(operand1, TRUE) * Int(operand2, TRUE));

X[d] = result<63:0>;
```

C6.2.281 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 for this instruction.

31	30 29	28 27 26 25 24	23	22 21	20 16	15	14 10	9 5	4 0
1	0 0	1 1 0 1 1	1	0 1	Rm	1	1 1 1 1 1	Rn	Rd
			U			o0	Ra		

64-bit variant

```
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.

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.282 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.

31	30 29	28 27 26 25 24	23	22 21	20 16	15	14 10	9 5	4 0
1	0 0	1 1 0 1 1	1	0 1	Rm	1	Ra	Rn	Rd
			U			o0			

64-bit variant

```
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

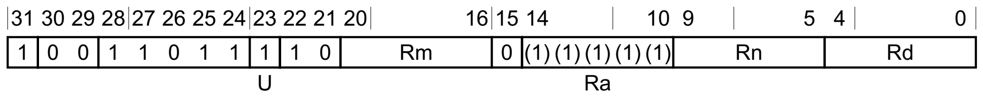
`<Xd>`	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
`<Wn>`	Is the 32-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
`<Wm>`	Is the 32-bit name of the second general-purpose source register holding 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];
bits(32) operand2 = X[m];
bits(64) operand3 = X[a];

integer result;

result = Int(operand3, TRUE) - (Int(operand1, TRUE) * Int(operand2, TRUE));
X[d] = result<63:0>;
```

C6.2.283 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.



64-bit variant

```
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];
bits(64) operand2 = X[m];

integer result;

result = Int(operand1, TRUE) * Int(operand2, TRUE);

X[d] = result<127:64>;
```

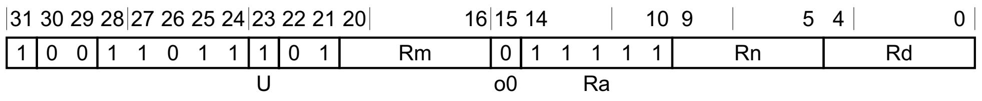
Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.284 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 for this instruction.



64-bit variant

```
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.

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.285 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 for this instruction.

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

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.

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.286 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 for this instruction.

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

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.

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.287 WFE

Wait For Event is a hint instruction that indicates that the PE can enter a low-power state and remain there until a wakeup event occurs. Wakeup events include the event signaled as a result of executing the SEV instruction on any PE in the multiprocessor system. For more information, see *Wait for Event mechanism and Send event* on page D1-1879.

As described in *Wait for Event mechanism and Send event* on page D1-1879, the execution of a WFE instruction that would otherwise cause entry to a low-power state can be trapped to a higher Exception level. See:

- *Traps to EL1 of EL0 execution of WFE and WFI instructions* on page D1-1845.
- *Traps to EL2 of Non-secure EL0 and EL1 execution of WFE and WFI instructions* on page D1-1861.
- *Traps to EL3 of EL2, EL1, and EL0 execution of WFE and WFI instructions* on page D1-1871.

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

System variant

```
WFE
```

Decode for this encoding

```
// Empty.
```

Operation

```
if IsEventRegisterSet() then
    ClearEventRegister();
else
    if PSTATE.EL == EL0 then
        // Check for traps described by the OS which may be EL1 or EL2.
        AArch64.CheckForWFxTrap(EL1, TRUE);
    if HaveEL(EL2) && !IsSecure() && PSTATE.EL IN {EL0, EL1} && !IsInHost() then
        // Check for traps described by the Hypervisor.
        AArch64.CheckForWFxTrap(EL2, TRUE);
    if HaveEL(EL3) && PSTATE.EL != EL3 then
        // Check for traps described by the Secure Monitor.
        AArch64.CheckForWFxTrap(EL3, TRUE);
    WaitForEvent();
```

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.288 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* on page D1-1882.

As described in *Wait For Interrupt* on page D1-1882, the execution of a `WFI` instruction that would otherwise cause entry to a low-power state can be trapped to a higher Exception level. See:

- *Traps to EL1 of EL0 execution of WFE and WFI instructions* on page D1-1845.
- *Traps to EL2 of Non-secure EL0 and EL1 execution of WFE and WFI instructions* on page D1-1861.
- *Traps to EL3 of EL2, EL1, and EL0 execution of WFE and WFI instructions* on page D1-1871.

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

System variant

```
WFI
```

Decode for this encoding

```
// Empty.
```

Operation

```
if !InterruptPending() then
    if PSTATE.EL == EL0 then
        // Check for traps described by the OS which may be EL1 or EL2.
        AArch64.CheckForWFxTrap(EL1, FALSE);
    if HaveEL(EL2) && !IsSecure() && PSTATE.EL IN {EL0, EL1} && !IsInHost() then
        // Check for traps described by the Hypervisor.
        AArch64.CheckForWFxTrap(EL2, FALSE);
    if HaveEL(EL3) && PSTATE.EL != EL3 then
        // Check for traps described by the Secure Monitor.
        AArch64.CheckForWFxTrap(EL3, FALSE);
    WaitForInterrupt();
```

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential

C6.2.289 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* on page B1-76.

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

System variant

```
YIELD
```

Decode for this encoding

```
// Empty.
```

Operation

```
Hint_Yield();
```

Copyright © 2013-2017 ARM Limited or its affiliates. All rights reserved.
Non-Confidential