Meta HTP

GP Technical Reference Manual - Instruction Set

Filename	:	Meta HTP.GP Technical Reference Manual - Instruction Set.doc
Version	:	2.1.235 External Issue HTP.
Issue Date	:	01 May 2013
Author	:	Imagination Technologies Limited

Contents

1. Introduction

This document describes the Meta general purpose (GP) instruction set in detail. For an overview of the Meta architecture and core register details please refer to the Architecture Overview TRM.

2. GP instruction set

This section describes the GP instructions implementation in assembler. Detailed implementation notes for each instruction group are given later in this section and corresponding encoding diagrams are given in section 3.

2.1. Assembler notation

Meta instructions are 32 bit. Assembler syntax is described in detail in the Meta Programmers Guide. A typical instruction takes the form:

```
Instruction Mnemonic[Modifiers]cc        Operands
```

`[Modifiers]` are made up of a set of characters that modify or affect the behaviour of the instruction.

`cc` are condition codes which are valid on all instructions that are conditional (support cc field).

2.1.1. Operand registers

Registers are represented by D0.r, D1.r, A0.r, A1.r, CT.r, TR.r, PC, PCX, TT.r, FX.r, CPC0, CPC1

Where:

D0.r and D1.r	is any data unit register.
A0.r and A1.r	is any address unit register.
CT.r, TR.r, PC, PCX	is control, trigger or PC unit registers.
TT.r, FX.r	is any trace unit or floating point unit register.
CPC0 and CPC1	is the value of the PC at the start of the instruction associated with address unit 0 and 1 respectively. *Note: They can only be used as source operands in the corresponding address unit.*

Most of these registers such as control registers have aliases (e.g. CT.0 is TXENABLE) which are given in the Architecture Overview TRM. The aliases are defined in header file machine.inc which is included in the Meta toolkit. Aliases or register numbers can be used in assembler code.

Extended notation

Because each data unit and their contents have similar capabilities the following notations are used in this document to document operations that are symmetrical (the same applies to the address units).

Dx | Ax	Any data | address unit
De.r | Ae.r	A register in the same data | address unit as Dx.r | Ax.r
Do.r | Ao.r	A register in the opposite data | address unit versus Dx.r | Ax.r
De.e | Ae.e	The same register in the same unit as Dx.r | Ax.r. Implies that source and destination operands must match to be valid.
Rx.r	A register in Do, Ax or RD data port.

BBx.r	A register in Dx or Ax (see also [BBa.r] in section 2.1.4 Address Modes)
BBe.r	A register in the same data | address unit as BBx.r (e.g. if BBx.r is D0.r, then BBe.r is D0.r)
BBo.r	A register in the opposite data | address unit as BBx.r (e.g. if BBx.r is D0.r, then BBo.r is D1.r)
BBo.e	The same register in the opposite data | address unit as BBx.r (e.g. if BBx.r is D0.1, then BBo.e is D1.1)
Ux.r	A register in Dx | Ax | RA | PC | TR | CT | TT
Uy.r	A register in Dx | Ax | RA | PC | TR | CT | TT . Implies that Uy must be a different unit from Ux.

Example:

```
ADD De.r, Dx.r, De.r
        |    |     |
        |    |     + ---- same unit as Dx.r
        |    + ---------- D0.r or D1.r
        + --------------- same unit as Dx.r

ADD D0.0, D0.1, D0.2   ;supported instruction
ADD D1.0, D0.1, D0.2   ;unsupported instruction (may still be
                        supported by another ADD pattern)
```

2.1.2. Read ports

Read ports are split into output RA (read address) and input RD (read data) elements. Extended read address ports are identified as:

RAPF	Read Address Prefetch
RABZ	Read Address Byte Zero Extend
RAWZ	Read Address Word Zero Extend
RADZ	Read Address Dword Zero Extend
RABX	Read Address Byte Sign Extend
RAWX	Read Address Word Sign Extend
RADX	Read Address Dword Sign Extend
RAM8X	Read Address MX Extend 8-bit
RAM16X	Read Address MX Extend 16-bit
RAM8X32	Read Address MX Extend 8x32

Extended notation

The notation RAxx is used in this document to indicate any one of the possible read address ports.

2.1.3. Coprocessor ports

Coprocessor ports are represented by CP.r

2.1.4. Address modes

Addresses are indicated by square brackets []. Any [Addr] used in the assembler can be represented by the following addressing methods where U denotes any unit.

Addr	**Comments**
U0.r+U0.r|U1.r+U1.r	Base offset by register [Note1]
U0.r++U0.r|U1.r++U1.r	Base pre-incremented by register offset [Note1]
U0.r+U0.r++|U1.r+U1.r++	Base post-incremented by register offset [Note1]
++U0.r|++U1.r|U0.r++|U1.r++	Base pre-incremented or post-incremented by Size
--U0.r|--U1.r|U0.r--|U1.r--	Base pre-decremented or post-decremented by Size
U0.r+#N|U1.r+#N	Base offset by constant
U0.r++#N|U1.r++#N	Base pre-incremented by constant offset
U0.r+#N++|U1.r+#N++	Base post-incremented by constant offset

Note 1: If memory store (SET) address expressions use an offset register then the data being written cannot be sourced from the same unit as the address; this affects dual-unit SETL operations most significantly as U0|U1 must be Ax if Dx data is stored and visa-versa.

The precise set of address expressions permitted can vary according to a number of factors. The shorthand to represent these permutations is shown below. Use of registers numbered 0 or 1 in any U0|U1 unit as the base register, will allow the largest range of constant offsets to be used. All constant offsets must be specified in bytes and be a multiple of the size of the operation concerned.

[BBa.r]	An address based on a register in Dx or Ax.
[BBa.r+#S6]	As above with a positive 6-bit signed immediate offset.
[BBxUaRoPp] [BBxUaS6Pp]	Can be interpreted as follows:

BBx	An address based on a register in Dx or Ax.
Ua	Optional post or pre-increment of the address base by the offset given.
Ro	Offset register in the same unit as BBx.
S6 | S8	Scaled signed 6-bit | 8-bit immediate offset.
Pp	Pre-increment is supported. When Ua is used without Pp only post increment is supported.

In general, all operations involving an address expression or the RD input port must have a size explicitly specified by the programmer to allow clear communication of the precise meaning of the operation.

When an address has been issued to RA*, the RD port can be used in many ALU operations as a direct parameter that implicitly uses the value retrieved from memory.

2.1.5. Instruction modifiers

The effects of the individual instruction modifiers are described below. The order top to bottom implies the order in which multiple modifiers can be concatenated:

Text	Type	Description
S	Set Condition Flags	See Condition-flag setting operations below.

Text	Type	Description
B|W|D|L	Size Flags	Size of transfer B - Byte Operation (8 bits) W - Word Operation (16 bits) D - DWord Operation (32 bits) L - LWord Operation (64 bits)
cc	Condition Code	TestCondition (EQ|Z|NE|NZ|CS|LO|CC|HS|MI|N|PL|NC|VS|VC|HI|LS|GE|LT|GT|LE)
T	Top Value	Parameter bits 31-16 are #Value16. Bits 15-0 are 0. Cannot be used with signed 16-bit immediates.
MT	Mask Top Value	Parameter bits 31-16 are #Value16. Bits 15-0 are 1.
MB	Mask Bottom Value	Parameter bits 15-0 are #Value16. Bits 31-16 are 1.
R	Repeat or Priority	See Repeating operations below. Indicates priority in ICACHE and ICACHEEXIT instructions.

2.1.6. Values

Any Value used in the assembler can contain a C-like expression made up of the following elements:

Text Literals	Description
[0-9]	Decimal constant; [] any number of char like those enclosed
0x[0-9][A-F] [a-f]	Hexadecimal constant
+, -, |, &, ^, *, /, %, <<, >>	Unary or dyadic C expression operators
==, !=, <, <=, >, >=, &&, ||	C logical expression operators; these yield the values 1 or 0
Label	Textual reference to a code or data label name. This is resolved during the linkage stage to the full 32-bit address of the code or data item concerned.

2.1.7. Immediate values

Immediate values may be used directly in a number of instructions. The general rule is that 16-bit signed or unsigned values may be used with most operations in a way similar to the ADD case; i.e. only one address or data unit register may be involved in the operation. Signed 16-bit immediates cannot make use of the T modifier.

Alternatively 8-bit unsigned values (0 to 255) may be used more flexibly in operations that involve independent source and destination registers.

Immediate values may be manipulated with the features of the Meta assembler input syntax below:

```
HI( )   High-Value     Extracts bits 31 to 16 of the value enclosed.
LO( )   Low-Value      Extracts bits 15 to 0 of the value enclosed.
```

In the instruction descriptions later in this section, immediate values are represented by the following notation:

Text	Description

Text	Description
#Snn	A signed offset, the value is always treated as signed.
#Inn	Signed or unsigned immediate, can be optionally sign extended.
#Xnn	Unsigned immediate, the value is always treated as unsigned.

2.1.8. Condition-flag setting operations

In order to support conditional operations controlled by the data unit, Meta supports operations that specifically exist to affect internal resources commonly known as condition-flags.

The following condition-flags exists in the Meta core:

- Zero flag - flag is set to indicate a zero result of an operation (no bits set)
- Negative flag - flag is set to indicate a negative result of an operation (bit 31 set)
- Overflow flag - flag is set to indicate the overflow/underflow of a signed operation
- Carry flag - flag is set to indicate the overflow/underflow of an unsigned operation or to indicate the state of the 'last bit discarded' during a shift operation

Instructions ABS, FFB, NORM, MIN, MAX, NMIN, CMP, TST, VPACK, MORT always set condition flags.

Lastly, but perhaps most importantly, it is possible to set the flags described above from the actual side effects and result of most other internal operations. This feature is requested by placing an S instruction modifier on the instruction concerned:

```
ALUS    Dest, Source1, Source2        ; ALU -> ADD, SUB, AND, OR, XOR
SHTS    Dest, Source1, Source2        ; SHT -> LSL, LSR, or ASR
MOVS    Dest, Source                  ; Source must be D0.r|D1.r
RTDSW   Dest, Source                  ; Source must be D0.r|D1.r
XSDSW   Dest, Source                  ; Source must be D0.r|D1.r
XSDSB   Dest, Source                  ; Source must be D0.r|D1.r
```

When combined with other instruction modifiers the S flag always comes first.

2.1.9. Conditional operations

A number of instructions can be executed conditionally on a Meta core. A conditional instruction is fetched, scheduled, and executed in normal sequence by the processor. However, any output or updating effects are only actually performed if the condition specified for the instruction is met.

The following cc modifiers specify the following conditions dependent on the operation used to set the flags (xxx R, A, B or xxx A, B):

Modifier	Meaning	CMP|SUB	TST|AND|OR|XOR|others …	ADD|MOV	LSL|LSR|ASR	Condition 'Flag' State
A	Always					
EQ|Z	Equal/Zero	A == B, R == 0	R == 0	R == 0		Zero == 1
NE|NZ	Not Equal/Not Zero	A != B, R != 0	R != 0	R != 0		Zero == 0
CS|LO	Carry Set/Lower	A $<_{unsigned}$ B		R$>=_{unsigned}$ 2^{32}	Nth bit lost == 1	**C**arry == 1
CC|HS	Carry Clear/Higher or Same	A $>=_{unsigned}$ B		R$<_{unsigned}$ 2^{32}	Nth bit lost == 0	**C**arry == 0
MI|N	Minus/N Set	R $<_{signed}$ 0	R < 0			Neg == 1
PL|NC	Positive/N Clear	R $>=_{signed}$ 0	R >= 0			Neg == 0
VS	Overflow Set	R_{signed} overflow				O**v**er == 1
VC	Overflow Clear	R_{signed} VALID				O**v**er == 0
HI	Higher (unsigned)	A $>_{unsigned}$ B				(**C**arry == 0) AND (**Z**ero == 0)
LS	Lower or Same (unsigned)	A $<=_{unsigned}$ B				(**C**arry == 1) OR (**Z**ero == 1)
GE	Greater Than or Equal	A $>=_{signed}$ B				**N**eg == O**v**er
LT	Less Than	A $<_{signed}$ B				**N**eg != O**v**er
GT	Greater Than	A $>_{signed}$ B				(**Z**ero == 0) AND (**N**eg == O**v**er)
LE	Less Than or Equal	A $<=_{signed}$ B				(**Z**ero == 1) OR (**N**eg != O**v**er)

The simplest form of conditional operation is a branch instruction Bcc.
The following more complex operations can also have a cc modifier applied to them:

```
        ALUcc   Dest, Source1, Source2          ;       ALU -> ADD, SUB, AND, OR, XOR, or MUL
                                                ;       Source2 immediate limited to 0-255

        BSHcc   Dest, Source1, Source2          ;       BSH -> LSL, LSR, or ASR bit-wise shifts

        MOVcc   Dest, Source                    ;       Almost any Dest or Source
                                                ;       can be supported.
                                                ;       Source immediate limited to 0-255

        SETxcc  [Dest], Source                  ;       Dest must be U0.r|U1.r
                                                ;       (no offset or adjustment)

        CMPcc   Source1, Source2                ;       Source2 immediate limited to 0-255

        TSTcc   Source1, Source2                ;       Source2 immediate limited to 0-255

        CPRxcc  Dest, CP.n                      ;       Data from co-processor is discarded
                                                ;       if condition is false

        CPRLcc  Dest1, Dest2, CP.n              ;       Data from co-processor is discarded
                                                ;       if condition is false
```

The cc modifier must be placed after any size specifying text as shown above and before any *T or R modifiers. If the RD port is used as a source parameter then data is required and consumed as in the co-processor read case whether or not the condition is met.

The instruction modifier S can also be specified as part of a conditional operation. In general this means that a conditional instruction will also set the thread's condition flags, but only if the specified condition is satisfied. For example:

```
        ADDSVS          D0Ar6, D0Ar6, D0Re0    ; Overflow condition triggers ADD and new flags
```

This instruction may be described as Add the two operands D0Ar6 and D0Re0 together. If the overflow is not already set, store the result in D0Ar6 and set the thread condition flags according to the result.

The cc test depends on a prior instruction, not the current one, for example:

```
        CMP             D0Re0, #0      ;(1)
        CMPEQ           D1Re0, #0      ;action depends upon (1)
        BEQ             ...            ;Branch if both D0Re0 and D1Re0 are zero
```

2.1.10. Repeating operations

The Meta core can execute the SIMD operations BR, XFR. CPW and CPXL a number of times dependant on the 32-bit limit value loaded into the control register, TXRPT. The TXRPT value is referred to as a limit value, since the value 0 in this register implies 1 iteration, whereas 0xFFFFFFFF implies 2^{32} iterations.

BR and XFR have their own encoding and are described separately, see section 2.2.7 and 2.6.30.

In the case of the coprocessor-write instructions CPW and CPXL, when the R instruction modifier is applied the instruction can be used in a repeating form to transfer a block of data to a coprocessor or between coprocessors, see sections 2.6.8 and 2.6.9.

```
        MOV     TXRPT, #6              ; Transfer eight 64-bit items to CP.n
        MOVL    RA, [SrcAddr]          ; Prefetch data for pipeline
        CPWR    CP.n, [SrcAddr]        ; Transfer seven 64-bit items to CP.n
        MOVL    D0Re0, D1Re0, RD       ; Read last data item into registers
        CPW     CP.n, D0Re0, D1Re0     ; Deliver last data item from registers
        MOV     TXRPT, #7       ; Set up the repeat counter
        CPXLR   CP.nd, CP.ns    ; Transfer eight 64-bit items from CP.ns to CP.nd
```

2.1.11. Symbolic register names in instruction examples

In some of the examples given in the instruction pages symbolic register names are used. The table below shows a summary of how the symbolic names map to core registers. A detailed explanation of calling conventions is given in the Meta and MTX GCC4 Compiler User Guide.

Register	Symbolic	Register	Symbolic
D0.0	D0Re0	D1.0	D1Re0
D0.1	D0Ar6	D1.1	D1Ar5
D0.2	D0Ar4	D1.2	D1Ar3
D0.3	D0Ar2	D1.3	D1Ar1
D0.4	D0FrT	D1.4	D1RtP
D0.5		D1.5	
D0.6		D1.6	
D0.7		D1.7	
D0.8			
A0.0	A0StP	A1.0	A1GbP
A0.1	A0FrP	A1.1	A1LbP
A0.2		A1.2	
A0.3		A1.3	

Note that where an instructions states that registers are passed in ascending order such as MSETx, ascending order means in register numerical order, i.e. D0.0, D0.1, D1.0, D1.0 etc., not in symbolic name order.

2.2. Core general purpose instructions

These instructions all have a short encoding format in common usage.

2.2.1. ADD Ae.r,Ax.r,Ae.r

Performs an internal address unit arithmetic ADD, can take the current PC as an input.
See encoding diagram 3.2.1

Syntax

ADD **Dst,Src1,Src2**

Dst	Ae.r
Src1	Ax.r | CPCx
Src2	Ae.r | CPCe

Operation

```
0:          'Dst   = AUAdd(Src1, Src2);
```

Examples

```
  ADD       A0.7,A0.7,A0FrP
  ADD       A1.7,A1LbP,CPC1
  ADD       A0FrP,CPC0,CPC0
```

Usage notes

When CPCn is specified, the relevant operand is the base address of the current instruction.

2.2.2. ADD Ae.r,Ax.r,#I16

Performs an internal address unit arithmetic ADD with 16-bit immediate data.

See encoding diagram 3.2.2

Syntax

ADD[T] **Dst,Src1,Src2**

Dst	Ae.r
Src1	Ax.r | CPCx
Src2	#I16

Operation

```
0:         IMMSrc2 = Src2,
    [T] -> IMMSrc2 <<= 16,
           'Dst    = AUAdd(Src1, IMMSrc2);
```

Examples

```
  ADD       A1.7,CPC1,#0x7fff              ; CPC1 relative address
  ADD       A0.7,A0.7,#0x7fff
  ADD       A0.7,CPC0,#-0x1
  ADDT      A0FrP,CPC0,#HI(0x7fff1234)     ; CPC0 + offset top 0x7FFF0000
  ADD       A0FrP,A0FrP,#LO(0x7fff1234)    ;  ... plus remainder 0x00001234
```

Usage notes

When CPCx is specified, the relevant operand takes the value of the current PC.

The `[T]` modifier selects bits 31 to 16 as those specified by the immediate operand; bits 15 to 0 are set to zero.

2.2.3. ADD De.r,Dx.r,De.r

Performs an internal data unit arithmetic ADD.

See encoding diagram 3.2.3

Syntax

ADD[S] **Dst,Src1,Src2**

Dst	De.r
Src1	Dx.r
Src2	De.r

Operation

```
0:         'Dst   = DUAdd(Src1, Src2),
    [S] -> 'Flags = DUFlags;
```

Examples

```
  ADD       D0.7,D0.7,D0Re0
  ADDS      D1.7,D1Re0,D1.7
```

Usage notes

2.2.4. ADD De.e,Dx.r,#I16

Performs an internal data unit arithmetic ADD with 16-bit immediate data.
See encoding diagram 3.2.4

Syntax

ADD[S][T] **Dst,Src1,Src2**

Dst	De.r
Src1	Dx.r
Src2	#I16

Operation

```
0:          IMMSrc2 = Src2,
     [T] -> IMMSrc2 <<= 16,
            'Dst    = DUAdd(Src1, IMMSrc2),
     [S] -> 'Flags  = DUFlags;
```

Examples

```
 ADDS D0.7,D0.7,#-0x8000            ;flag setting ADD
 ADDT D0.7,D0.7,#HI(0x7fff1234)     ;ADD top half of constant   0x7FFF0000
 ADD  D0.7,D0.7,#LO(0x7fff1234)     ;  ...   and then remainder 0x00001234
```

Usage notes

The **`[T]`** modifier selects bits 31 to 16 as those specified by #I16. Bits 15 to 0 set to zero.

2.2.5. AND De.r,Dx.r,De.r

Performs a bitwise AND of two operands and stores the result in the source data unit.

See encoding diagram 3.2.5

Syntax

AND[S] **Dst,Src1,Src2**

Dst	De.r
Src1	Dx.r
Src2	De.r

Operation

```
0:         'Dst   = DUAnd(Src1, Src2),
   [S] ->  'Flags = DUFlags;
```

Examples

```
  AND       D0.7,D0.7,D0.7
  ANDS      D1Re0,D1.7,D1.7
```

Usage notes

2.2.6. AND De.e,Dx.r,#I16

Perform a bitwise AND with 16-bit immediate data in the data units using the same register for both source and destination.

See encoding diagram 3.2.6

Syntax

AND[S][T|MT|MB] **Dst,Src1,Src2**

Dst	De.e
Src1	Dx.r
Src2	#I16

Operation

```
0:           IMMSrc2 = Src2,
    [T]  -> IMMSrc2 <<= 16,
    [MT] -> IMMSrc2 |= 0x0000FFFF,
    [MB] -> IMMSrc2 |= 0xFFFF0000,
            'Dst    = DUAnd(Src1, IMMSrc2),
    [S]  -> 'Flags  = DUFlags;
```

Examples

```
  AND       D0Re0,D0Re0,#-0x8000          ; AND with 0xFFFF8000
  ANDT      D1Re0,D1Re0,#0x7fff           ; AND with 0x7fff0000
  ANDMT     D0.7,D0.7,#HI(0x7fff1234)     ; AND with 0x7fffffff ...
  ANDMB     D0.7,D0.7,#LO(0x7fff1234)     ;  ... then 0xffff1234
```

Usage notes

The **`[T]`** or Top modifier selects bits 31 to 16 as those specified by #I16. Bits 15 to 0 set to zero.

The **`[MT]`** or Mask Top modifier is similar to the T case except bits 15 to 0 are set to one.

The **`[MB]`** or Mask Bottom modifier sets bits 15 to 0 from #I16 with bits 31 to 16 set to one.

2.2.7. B #S19

Performs a jump by a number of 32-bit word units or to a label.

See encoding diagram 3.2.7

Syntax

B[R]	**Dst**
Dst	#S19 | Label

Operation

B

```
0:                       'PC = PCUAdd(PC, Dst);
```

BR

```
0:   (TXRPT != 0) ->     'PC = PCUAdd(PC, Dst),
     (TXRPT != 0) -> 'TXRPT = TXRPT - 1;
```

Examples

```
  B    #+262143                 ; Maximum forward jump offset
  B    Label                    ; Jump to label
  BR   #Label                   ; Loop controlled by TXRPT--
```

Usage notes

The branch instruction takes a 19-bit signed offset (-262144 <= Value < 262144) in terms of 32-bit instructions (i.e. the offset undergoes a fixed multiply by four before use), which gives a complete offset range of -1048576 to +1048572 bytes from the base of the branch instruction.

An offset of 0 generates a single instruction loop (branch to self).

The repeat modifier `[R]` is generally used to construct counted loops:

```
MOV     TXRPT, #6      ; Loop will be executed seven times!
Loop:                  ;
                       ; Contents of loop
                       ;
BR      Loop           ; Jump will occur six times!
```

2.2.8. Bcc #S19

Performs a conditional jump by a number of instructions or to a label.

See encoding diagram 3.2.8

Syntax

Bcc[R] **Dst**

Dst #S19 | Label

Operation

B

```
0: <cc> ->   'PC = PCUAdd(PC, Dst);
```

BR

```
0:  <cc> && (TXRPT != 0) ->      'PC = PCUAdd(PC + Dst),
            (TXRPT != 0) -> __'TXRPT = TXRPT - 1;
```

Examples

```
BGE       #-1
BLTR      #+262143
BPL       Label
```

Usage notes

The branch instruction takes a 19-bit signed offset (-262144 <= Value < 262144) in terms of 32-bit instructions (i.e. the offset undergoes a fixed multiply by four before use), which gives a complete offset range of -1048576 to +1048572 bytes from the base of the branch instruction.

An offset of 0 generates a single instruction loop (branch to self).

Use of conditional instructions is preferable to short conditional branches if this is practical given the nature of the code being skipped by the jump.

The repeat modifier **`[R]`** is generally used to construct counted loops as shown in the non-conditional branch case above. In more complex loops, a terminating condition can be added to short-cut the execution when required:

```
MOV     TXRPT, #6      ; Loop may be executed seven times!
Loop:                  ;
                       ; Contents of loop

CMP     Data, #0       ; Terminate loop early if Data is zero
BNZR    Loop           ; Jump may occur six times or less!
```

This causes the instruction to make the branch decision based not only upon the condition code but also the value contained in the repeat counter. The rule used is that if the condition code is satisfied AND the repeat counter is not zero the branch is taken, otherwise it is not taken. Regardless of whether the branch is taken or not, the repeat counter is decremented towards zero (but not beyond). A common use of the loop feature is to reduce the cost of a counting down loop from two instructions (CMP and Bcc) to just one. This type of use is by setting the cc field to the "always" condition. If the "never" condition is used this instruction will not repeat, although the repeat counter may be decremented.

A simple non-repeat branch which uses a "never" (NV) condition code is effectively a NOP.

See section 2.1.9 Conditional operations.

2.2.9. CALLR BBx.r,#S19

Performs a relative call to a sub-routine by an offset number of instructions.

See encoding diagram 3.2.9

Syntax

CALLR **Link,Value**

Link	Dx.r | Ax.r
Value	#S19 | Label

Operation

```
0:   'Link = PCUNext(PC),
     'PC   = PCUAdd(PC, Value);
```

Examples

```
  CALLR     D1RtP,Label               ; Normal relative sub-routine call
  CALLR     D1.0,#16                  ; Jump to subsequent code with Link
```

Usage notes

The relative call offset is in terms of 32-bit instructions (i.e. the immediate undergoes a fixed multiply by four before use), which gives a complete offset range of -1048576 to +1048572 bytes.

After issuing the instruction the PC is set to current PC +/- the offset and **`Link`** will have the correct return address written to it (Link register = InstructionAddr + InstructionSize).

2.2.10. CMP Dx.r,De.r

Updates all the condition flags to indicate the result of an equivalent subtract operation based on the two input parameters.

See encoding diagram 3.2.10

Syntax

CMP	**Src1,Src2**
Src1	Dx.r
Src2	De.r

Operation

```
0:        DUSub(Src1, Src2),
          'Flags    = DUFlags;
```

Examples

```
  CMP       D0.7,D0.7
  CMP       D1Re0,D1.7
```

Usage notes

See section 2.1.8 Condition-flag setting operations.

2.2.11. CMP Dx.r,#I16

Updates all the condition flags to indicate the result of an equivalent subtract operation based on the two input parameters.

See encoding diagram 3.2.11

Syntax

CMP[T|MT|MB] **Src1,Src2**

Src1	Dx.r
Src2	#I16

Operation

```
0:          IMMSrc2 = Src2,
   [T]  -> IMMSrc2 <<= 16,
   [MT] -> IMMSrc2 |= 0x0000FFFF,
   [MB] -> IMMSrc2 |= 0xFFFF0000,
           DUSub(Src1, IMMSrc2),
           'Flags  = DUFlags;
```

Examples

```
  CMP       D0.7,#-0x1
  CMPT      D1Re0,#-0x8000
```

Usage notes

See section 2.1.8 Condition-flag setting operations.

The **`[T]`** or Top modifier selects bits 31 to 16 as those specified by #I16. Bits 15 to 0 set to zero.

The **`[MT]`** or Mask Top modifier is similar to the T case except bits 15 to 0 are set to one.

The **`[MB]`** or Mask Bottom modifier sets bits 15 to 0 from #I16 with bits 31 to 16 set to one.

2.2.12. CMP Dx.r,Rx.r

Updates all the condition flags to indicate the result of an equivalent subtract operation based on the two input parameters.

See encoding diagram 3.2.12

Syntax

CMP **Src1,Src2**

Src1	Dx.r
Src2	Dx.r | Ax.r | RD

Operation

```
-1:       O2RSrc2   = O2R(Src2);
0:        DUSub(Src1, O2RSrc2),
          'Flags    = DUFlags;
```

Examples

```
  CMP       D0.7,A1.7
  CMP       D0Re0,D1.7  ;
  CMP       D0Re0,RD
```

Usage notes

See section 2.1.8 Condition-flag setting operations.

See Operand 2 replace in section 3.1.3

2.2.13. GETD BBx.r, [A0.r+#S3]

Loads a single register from using an A0 unit sourced address and a signed 3-bit 32-bit word offset.
See encoding diagram 3.2.13

Syntax

GET[B|W|D] **Dst,[Src]**

Dst	BBx.r
[Src]	[A0.r+#S3]

Operation

```
0:          Addr = BUAddr(Src),
            Size = D;
2:          Data = MemLoad(Addr, Size);
3:          'Dst = Data;
```

Examples

```
  GETD      A0FrP,[A0StP+#-0x10]
  GETD      D1Re0,[A0StP+#-0x4]
  GETD      A0.2,[A0FrP+#-0x8]
  GETD      A1.2,[A0FrP+#0xC]
  GETD      D1Re0,[A0.3]
```

Usage notes

To optimise this instruction, the Meta core attempts to execute subsequent instructions for the same thread while single GET reads are outstanding. Subsequent use of the destination register of the GET will force the processor to wait for the data concerned if has not already arrived.

See section 2.1.4 Address modes.

2.2.14. LOCKn

Supports mutual exclusion between the threads for some critical operations to be implemented.

See encoding diagram 3.2.14

Syntax

LOCKn

Operation

Examples

```
LOCK0
LOCK1
LOCK2
```

Usage notes

LOCK2 - Requests both the global and voluntary exclusion interlocks, when the global exclusion is granted it prevents all other threads for executing any other instructions of any type.

LOCK1 - Request the voluntary exclusion and prevent other threads from completing a LOCK1 (or LOCK2) instruction until the voluntary exclusion is released.

LOCK0 - Release all exclusions (voluntary and global), this operation can be executed even if no exclusion is currently held, but then it only functions as a NOP.

The use of the LOCK instruction will be interlocked until no outstanding memory transactions exist for the given thread. If a thread has any outstanding pipelined read requests when attempting to issue this instruction that thread will be halted and a suitable trigger will be issued (both internally and externally to the global trigger matrix).

See Privilege/Lock section in the Architecture Overview TRM.

2.2.15. MGET BBx.r..BBe.r..,[BBa.r]

Loads a set of up to eight registers in a single unit from memory.

See encoding diagram 3.2.15

Syntax

MGET[D|L] **RegList,[Src]**

RegList	BBx.r….BBe.r	
[Src]	[BBa.r]	(*restricted to Dx.(0-15) and Ax.(0-7))

Operation

```
0+n:             Addr$_n$   = BUAddr(Src),
                 Size    = [D|L];
2+n:             Data$_n$   = MemLoad(Addr$_n$, Size);
3+n:             'Dst$_n$   = Data$_n$;
        [L] ->   'Dst$_n$.o = Data$_n$.o;
```

Examples

```
  MGETD     A0FrP,A0.7,[A1.7++]
  MGETD     A0FrP,A0.7,[D0.7++]
  MGETD     D1Re0,D1.7,[A1.7++]
  MGETL     D0Re0,D0.7,[A0.7++]
```

Usage notes

*Global (Data and Address) registers are not accessible by these instructions.

RegList is a comma separated list that must contain at least two registers ((rmax-rmin) < 8). Specified order is ignored - registers are loaded in ascending order, i.e D0.0, D0.1, D1.0, D1.1 etc. The unit specified in RegList can be the same as the unit providing the load address, but the register providing the load address is NOT in the RegList.

`[D|L]` - [BBa.r] is post incremented by size D or L. The "++" post-increment operator is optional in the assembler source as it is implied for all MGET operations.

`[L]` - Loads up to eight pairs of similarly numbered registers from a pair of units (with D0/D1 and A0/A1 being the allowed pairs). In this case the pointer must be 64-bit aligned. If the unit specified in RegList is a data unit, then only an address unit register can be used for the address and visa-versa. The registers actually specified will be loaded first as the least significant bits of each 64-bit value generated.e.g. in the case of :

```
        MGETL     D0Re0,D0.7,[A0.7++]
```

the actual order of the registers loaded is:

```
        D0Re0,D1Re0,D0.7,D1.7
        i.e. D0.0,D1.0,D0.7,D1.7
```

2.2.16. MOV Ae.r,Ax.r

Performs an internal address unit move, can take the current PC as an input.
See encoding diagram 3.2.16

Syntax

MOV **Dst,Src**

Dst	Ae.r
Src	Ax.r | CPCx

Operation

```
0:          'Dst = AUMov(Src);
```

Examples

```
  MOV       A0.7,A0.7
  MOV       A1.7,A1.7
  MOV       A0FrP,CPC0
  MOV       A1LbP,CPC1
```

Usage notes

When CPCn is specified, Src takes the value of the current PC.

2.2.17. MOV Ax.r,#I16

Performs an internal address unit move with 16-bit immediate data.
See encoding diagram 3.2.17

Syntax

MOV[T]	**Dst,Src**
Dst	Ax.r
Src	#I16

Operation

```
0:            IMMSrc = Src,
      [T] -> IMMSrc <<= 16,
            'Dst    = AUMov(IMMSrc);
```

Examples

```
  MOV       A0.7,#-0x1
  MOV       A1.7,#-0x1
  MOVT      A0.7,#0x7fff
```

Usage notes

The **`[T]`** modifier selects bits 31 to 16 as those specified by #I16; bits 15 to 0 are set to zero.

2.2.18. MOV De.r,Dx.r

Performs an internal data unit move.

See encoding diagram 3.2.18

Syntax

MOV[S] **Dst,Src**

Dst	De.r
Src	Dx.r

Operation

```
0:          'Dst   = DUMov(Src1, Src2),
    [S] -> 'Flags = DUFlags;
```

Examples

```
  MOV       D0.7,D0.7
  MOVS      D1.7,D1Re0
```

Usage notes

2.2.19. MOV Dx.r,#I16

Performs an internal data unit move with 16-bit immediate data.

See encoding diagram 3.2.19

Syntax

MOV[S][T] **Dst,Src**

Dst	Dx.r
Src	#I16

Operation

```
0:          IMMSrc   = Src
    [T] ->  IMMSrc   <<= 16,
            'Dst     = DUMov(IMMSrc),
    [S] ->  'Flags   = DUFlags;
```

Examples

```
  MOV       D0Re0,#-0x1
  MOVS      D1.7,#0x7fff
  MOVST     D1Re0,#0x7fff
  MOVT      D0.7,#0x8000
```

Usage notes

The `[T]` modifier selects bits 31 to 16 as those specified by #I16. Bits 15 to 0 set to zero.

2.2.20. MOV Ux.r,Uy.r

Performs a cross-unit move.

See encoding diagram 3.2.20

Syntax

MOV **Dst,Src**

Dst	Do.r | Ao.r | PC.r | CT.r |TR.r | TT.r | RA.x
Src	Dx.r | Ax.r | PC.r | CT.r | TR.r | TT.r

Operation

```
0:          XSrc = XUMov(Src),
1:          'Dst = XSrc;
```

Examples

```
  MOV A0.1,D0.7
  MOV A0.7 D1.1
  MOV A1.1,D1.1
  MOV D0.r,A0.r
  MOV PC,D1.3
  MOV PC,A0.7
  MOV RAMX,A0.1
```

Usage notes

When Dst is RA.x, Src is restricted to Dx.r | Ax.r. This form allows the value of a register to be issued as an address into the read pipeline:

```
  MOV   RA*, Src              ; RA* -> RA|RABX|RABZ|RAWX|RAWZ|RADX|RADZ|RAM8X|RAM8X32
```

2.2.21. MSETx [BBa.r],BBx.r..BBe.r..

Stores a set of up to eight registers in a single unit to memory.

See encoding diagram 3.2.21

Syntax

MSET[D|L] **[Dst],RegList**

[Dst]	[BBa.r]	
RegList	BBx.r….BBe.r	(*restricted to Dx.(0-15) and Ax.(0-7))

Operation

```
0+n:           Addr_n   = BUAddr(Dst),
               Size     = [D|L],
               Data_n   = Src_n,
       [L] ->  Data_n.o = Src_n.o;
2+n:           MemStore(Addr_n, Size, Data_n.o:Data_n);
```

Examples

```
MSETD    [A0.7++],A0FrP,A0.2
MSETD    [A1.7++],D0Re0,D0.3
MSETD    [D0Re0++],A1LbP,A1.3
MSETL    [A0.7++],A1LbP,A1.3
MSETL    [A0.7++],D0Re0,D0.7
MSETL    [D0.7++],A0FrP,A0.3
MSETL    [D1.7++],A0FrP,A0.3
MSETL    [D1.7++],A1LbP,A1.3
MSETL    [D1.7++],D0Re0,D0.7
MSETL    [D1Re0++],D1Re0,D1.7
```

Usage notes

*Global (Data and Address) registers are not accessible by these instructions.

RegList is a comma separated list that must contain at least two registers ((rmax-rmin) < 8). Specified order is ignored - registers are stored in ascending order, i.e D0.0, D0.1, D1.0, D1.1 etc. .

`[D|L]` - [BBa.r] is post incremented by size D or L. The "++" post-increment operator is optional within the assembler source as it is implied for all MSET operations.

`[L]` - Stores up to eight pairs of similarly numbered registers from a pair of units by using the 64-bit form of the instruction (with D0/D1 and A0/A1 being the allowed pairs). In this case the pointer must be 64-bit aligned and if data unit registers are being saved then only an address unit register can be used for the address and visa-versa. Registers are stored in ascending order, with the unit specified being stored first.

e.g. in the case of :

```
MSETL [A0StP], D1Ar3,D1Ar1
```

the actual order of the registers stored is:

```
D1Ar3,D0Ar4,D1Ar1,D0Ar2
i.e. D1.2,D0.2,D1.3,D0.3
```

2.2.22. MUL De.r,Dx.r,De.r

Performs an internal data unit multiply.

See encoding diagram 3.2.22

Syntax

MUL[D|W] **Dst,Src1,Src2**

Dst	De.r
Src1	Dx.r
Src1	De.r

Operation

```
0:          MULRes = DUMul(Src1, Src2);
2:          'Dst   = MULRes;
```

Examples

```
  MULD      D1Re0,D1Re0,D1.7
  MULD      D1Re0,D1Re0,D1Re0
  MULW      D0.7,D0.7,D0.7
  MULW      D0.7,D0.7,D0Re0
```

Usage notes

MULW performs an unsigned 16 x 16 multiply which takes multiple cycles. The returned value is the full 32-bit result.

MULD is an unsigned 32 x 32 multiply which takes multiple cycles. The returned value is the lower 32-bits of the 64-bit result.

2.2.23. MUL De.r,Dx.r,#I16

Performs a data unit multiply with 16-bit immediate data.

See encoding diagram 3.2.23

Syntax

MUL[D|W][T] **Dst,Src1,Src2**

Dst	De.r
Src1	Dx.r
Src1	#I16

Operation

```
0:          IMMSrc2 = Src2,
     [T] -> IMMSrc2 <<= 16,
            MULRes  = DUMul(Src1, IMMSrc2);
2:          'Dst    = MULRes;
```

Examples

```
  MULD      D1.7,D1.7,#0x7fff
  MULDT     D1Re0,D1Re0,#0xffff
  MULW      D0.7,D0.7,#0x7fff
  MULWT     D1Re0,D1Re0,#0x8000
```

Usage notes

MULW performs an unsigned 16 x 16 multiply which takes multiple cycles. The returned value is the full 32-bit result.

MULD is an unsigned 32 x 32 multiply which takes multiple cycles. The returned value is the lower 32-bits of the 64-bit result.

The **`[T]`** modifier selects bits 31 to 16 as those specified by #I16. Bits 15 to 0 set to zero.

2.2.24. NEG Ae.r,Ax.r

Performs an internal address unit negation, can take the current PC as an input.
See encoding diagram 3.2.24

Syntax

NEG	**Dst,Src**
Dst	Ae.r
Src	Ax.r | CPCx

Operation

```
0:          'Dst = AUNeg(Src);
```

Examples

```
  NEG       A0.7,A0.7
  NEG       A1.7,A1.7
  NEG       A0FrP,CPC0
  NEG       A1LbP,CPC1
```

Usage notes

When CPCn is specified, Src takes the value of the current PC.

2.2.25. NEG Ax.r,#I16

Performs an internal address unit negation with 16-bit immediate data.
See encoding diagram 3.2.25

Syntax

NEG[T]	**Dst,Src**
Dst	Ax.r
Src	#I16

Operation

```
0:           IMMSrc = Src,
      [T] -> IMMSrc <<= 16,
            'Dst    = AUNeg(IMMSrc);
```

Examples

```
  NEG       A0.7,#-0x1
  NEG       A1.7,#-0x1
  NEGT      A0.7,#0x7fff
```

Usage notes

The **`[T]`** modifier selects bits 31 to 16 as those specified by #I16; bits 15 to 0 are set to zero.

2.2.26. NEG De.r,Dx.r

Performs an internal data unit negation.

See encoding diagram 3.2.26

Syntax

NEG[S] **Dst,Src**

Dst	De.r
Src	Dx.r

Operation

```
0:          'Dst   = DUNeg(Src1, Src2),
    [S] -> 'Flags = DUFlags;
```

Examples

```
  NEG       D0.7,D0.7
  NEGS      D1.7,D1Re0
```

Usage notes

2.2.27. NEG Dx.r,#I16

Performs an internal data unit negation with 16-bit immediate data.
See encoding diagram 3.2.27

Syntax

NEG[S][T] **Dst,Src**

Dst	Dx.r
Src	#I16

Operation

```
0:          IMMSrc   = Src
    [T] ->  IMMSrc   <<= 16,
            'Dst     = DUNeg(IMMSrc),
    [S] ->  'Flags   = DUFlags;
```

Examples

```
  NEG       D0Re0,#-0x1
  NEGS      D1.7,#0x7fff
  NEGST     D1Re0,#0x7fff
  NEGT      D0.7,#0x8000
```

Usage notes

The **`[T]`** modifier selects bits 31 to 16 as those specified by #I16. Bits 15 to 0 set to zero.

2.2.28. NOP

An instruction with no effect.

See encoding diagram 3.2.28

Syntax

NOP

Operation

```
0:          ;
```

Examples

```
  NOP
```

Usage notes

This operation may be used to pad code and waste time. The operation performed does not use any significant resources.

2.2.29. OR De.r,Dx.r,De.r

Performs a bitwise OR of two operands and stores the result in the source data unit.

See encoding diagram 3.2.29

Syntax

OR[S] **Dst,Src1,Src2**

Dst	De.r
Src1	Dx.r
Src2	De.r

Operation

```
0:          'Dst   = DUOr(Src1, Src2),
   [S] ->   'Flags = DUFlags;
```

Examples

```
  OR       D0.7,D0.7,D0.7
  ORS      D1Re0,D1.7,D1.7
```

Usage notes

2.2.30. OR De.e,Dx.r,#I16

Perform a bitwise OR with 16-bit immediate data in the data units using a similar register for both source or destination.

See encoding diagram 3.2.30

Syntax

OR[S][T|MT|MB] **Dst,Src1,Src2**

Dst	De.r
Src1	Dx.r
Src2	#I16

Operation

```
0:          IMMSrc2 = Src2,
    [T]  -> IMMSrc2 <<= 16,
    [MT] -> IMMSrc2 |= 0x0000FFFF,
    [MB] -> IMMSrc2 |= 0xFFFF0000,
            'Dst    = DUOr(Src1, IMMSrc2),
    [S]  -> 'Flags  = DUFlags;
```

Examples

```
  OR      D0Re0,D0Re0,#-0x8000          ; OR with 0xFFFF8000
  ORT     D1Re0,D1Re0,#0x7fff           ; OR with 0x7fff0000
  ORMT    D0.7,D0.7,#HI(0x7fff1234)     ; OR with 0x7fffffff ...
  ORMB    D0.7,D0.7,#LO(0x7fff1234)    ;  ... then 0xffff1234
```

Usage notes

The **`[T]`** or Top modifier selects bits 31 to 16 as those specified by #I16. Bits 15 to 0 set to zero.

The **`[MT]`** or Mask Top modifier is similar to the T case except bits 15 to 0 are set to one.

The **`[MB]`** or Mask Bottom modifier sets bits 15 to 0 from #I16 with bits 31 to 16 set to one.

2.2.31. RTH

Switches between interrupt level and background level when handling an interrupt.

See encoding diagram 3.2.31

Syntax

RTH

Operation

```
0:      'TXSTATUS.ISTAT = ISTAT ^ 1;
```

Examples

```
  RTH
```

Usage notes

RTH instruction toggles the interrupt level flag (ISTAT) to switch the thread between background level and interrupt level.

The use of the RTH instruction is interlocked until no outstanding memory transactions exist for the given thread. If a thread has any outstanding pipelined read requests when attempting to issue this instruction that thread will be halted and a suitable trigger will be issued (both internally and externally to the global trigger matrix).

2.2.32. RTI

Return from interrupts.

See encoding diagram 3.2.32

Syntax

RTI

Operation

```
0:      'TXSTATUS.ISTAT = 0,
        'PC             = PCX,
        'PCX            = PCUNext(PC);
```

Examples

```
  RTI
```

Usage notes

The interrupt level flag ISTAT is cleared and the thread is restored to background operation (swap PC and PCX). However, before returning from the interrupt, the interrupt PC must be set to the entry point for the interrupt handler. This results in the following code fragment:

```
ITerminate:
               RTI    ; Return from interrupt
IEntryPoint:
               ;
               ; Interrupt handler code
               ;
               B      ITerminate
```

The use of the RTI instruction is interlocked until no outstanding memory transactions exist for the given thread. If a thread has any outstanding pipelined read requests when attempting to issue this instruction that thread will be halted and a suitable trigger will be issued (both internally and externally to the global trigger matrix).

2.2.33. SETD [A0.r+#S3],BBx.r

Store a single register to memory using an A0 unit sourced address with a signed 3-bit 32-bit word offset.

See encoding diagram 3.2.33

Syntax

SET[B|W|D] **[Dst],Src**

[Dst]	[A0.r+#S3]
Src	BBx.r

Operation

```
0:          Addr   = BUAddr(Dst),
            Size   = D,
            Data   = Src;
2:          MemStore(Addr, Size, Data);
```

Examples

```
  SETD      [A0StP++#-0x4],D0Re0
  SETD      [A0StP+#-0x10],A0.2
  SETD      [A0FrP+#0xC],A1.2
  SETD      [A0.3],D1Re0
```

Usage notes

The same address/data unit may be used to generate the memory address and specify the register to be loaded.

See section 2.1.4 Address modes.

2.2.34. SUB Ae.r,Ax.r,Ae.r

Performs an internal address unit arithmetic subtraction, can take the current PC as an input.

See encoding diagram 3.2.34

Syntax

SUB	**Dst,Src1,Src2**
Dst	Ae.r
Src1	Ax.r | CPCx
Src2	Ae.r | CPCe

Operation

```
0:          'Dst   = AUSub(Src1, Src2);
```

Examples

```
  SUB       A0.7,A0.7,A0FrP
  SUB       A1.7,A1LbP,CPC1
```

Usage notes

When CPCn is specified, the relevant operand is the base address of the current instruction.

2.2.35. SUB Ae.r,Ax.r,#I16

Performs an internal address unit arithmetic subtraction with 16-bit immediate data.

See encoding diagram 3.2.35

Syntax

SUB[T] **Dst,Src1,Src2**

Dst	Ae.r
Src1	Ax.r | CPCx
Src2	#I16

Operation

```
0:         IMMSrc2 = Src2,
    [T] -> IMMSrc2 <<= 16,
           'Dst    = AUSub(Src1, IMMSrc2);
```

Examples

```
  SUB       A1.7,CPC1,#0x7fff               ; CPC1 relative address
  SUB       A0.7,A0.7,#0x7fff
  SUB       A0.7,CPC0,#-0x1
  SUBT      A0FrP,CPC0,#HI(0x7fff1234)     ; CPC0 + offset top 0x7FFF0000
  SUB       A0FrP,A0FrP,#LO(0x7fff1234)    ;  ... plus remainder 0x00001234
```

Usage notes

When CPCx is specified, the relevant operand takes the value of the current PC.

The `[T]` modifier selects bits 31 to 16 as those specified by the immediate operand; bits 15 to 0 are set to zero.

2.2.36. SUB De.r,Dx.r,De.r

Performs an internal data unit arithmetic subtraction.

See encoding diagram 3.2.36

Syntax

SUB[S] **Dst,Src1,Src2**

Dst	De.r
Src1	Dx.r
Src2	De.r

Operation

```
0:          'Dst   = DUSub(Src1, Src2),
    [S] -> 'Flags = DUFlags;
```

Examples

```
  SUB       D0.7,D0.7,D0Re0
  SUBS      D1.7,D1Re0,D1.7
```

Usage notes

2.2.37. SUB De.e,Dx.r,#I16

Performs an internal data unit arithmetic subtraction with 16-bit immediate data.
See encoding diagram 3.2.37

Syntax

SUB[S][T] **Dst,Src1,Src2**

Dst	De.r
Src1	Dx.r
Src2	#I16

Operation

```
0:          IMMSrc2 = Src2,
     [T] -> IMMSrc2 <<= 16,
            'Dst    = DUSub(Src1, IMMSrc2),
     [S] -> 'Flags  = DUFlags;
```

Examples

```
 SUBS D0.7,D0.7,#-0x8000            ;flag setting SUB
 SUBT D0.7,D0.7,#HI(0x7fff1234)     ;SUB top half of constant   0x7FFF0000
 SUB  D0.7,D0.7,#LO(0x7fff1234)     ;  ...   and then remainder 0x00001234
```

Usage notes

The **`[T]`** modifier selects bits 31 to 16 as those specified by #I16. Bits 15 to 0 set to zero.

2.2.38. SWAP Ux.r,Uy.r

Performs a simultaneous transfer between internal processor locations in different units.

See encoding diagram 3.2.38

Syntax

SWAP **Dst,Src**

Dst	BBo.r | PC.r | CT.r | TR.r | TT.r
Src	BBx.r | PC.r | CT.r | TR.r | TT.r

Operation

```
0:          SWAPSrc = Src,
            SWAPDst = Dst;
1:          'Dst    = SWAPSrc,
            'Src    = SWAPDst;
```

Examples

```
  SWAP      PC,D0.1
  SWAP      A0.7,D0.1
  SWAP      D0.1,TXMASKI
  SWAP      D0.1,PCX
```

Usage notes

Dst and Src must be in different units.

SWAP PC, Ux.r effectively implements a call 'to address value in register'.

PC.r | CT.r | TR.r | TT.r must be treated as if they are a single unit. So it is not possible to SWAP within this set of units e.g. SWAP CT.r, PC.r is not allowed.

2.2.39. SWITCH #-1|0xC3020E|0xFF3FFF

Represents an undefined operation that is not executed, but causes an exception that either triggers pre-configured exception handling code to execute or causes the thread to enter a HALT state.

See encoding diagram 3.2.39

Syntax

SWITCH **#Value24**

Operation

This instruction is never executed.

Examples

```
SWITCH          #0xffffff
```

Usage notes

This instruction is useful, for example, to place breakpoints in executable code for debugging purposes. The SWITCH instruction allows this situation to be differentiated from a thread stopping because an unknown instruction has been encountered.

The SWITCH instruction is not actually executed and hence the PC does not advance. This is useful for removing breakpoints from code and replacing them with the original instruction, as the current PC for the thread does not have to be wound back before restarting execution.

Value24 is a completely arbitrary value that can be used to implement software traps. The value can be extracted easily from the instruction and used by a trigger/interrupt handler to make decisions about further processing.

2.2.40. TST Dx.r,De.r

Updates all the condition flags to indicate the result of an equivalent bit-wise AND operation based on the two input parameters.

See encoding diagram 3.2.40

Syntax

TST	**Src1,Src2**
Src1	Dx.r
Src2	De.r

Operation

```
0:        DUAnd(Src1, Src2),
          'Flags    = DUFlags;
```

Examples

```
  TST       D0.7,D0.7
  TST       D1Re0,D1.7
```

Usage notes

See section 2.1.8 Condition-flag setting operations.

2.2.41. TST Dx.r,#I16

Updates all the condition flags to indicate the result of an equivalent bit-wise AND operation based on the two input parameters.

Syntax

TST[T|MT|MB] **Src1,Src2**

Src1	Dx.r
Src2	#I16

Operation

```
0:         IMMSrc2 = Src2,
   [T]  -> IMMSrc2 <<= 16,
   [MT] -> IMMSrc2 |= 0x0000FFFF,
   [MB] -> IMMSrc2 |= 0xFFFF0000,
           DUAnd(Src1, IMMSrc2),
           'Flags  = DUFlags;
```

Examples

```
  TST       D0.7, ,#-0x1
  TSTT      D1Re0, #-0x8000
```

Usage notes

See section 2.1.8 Condition-flag setting operations.

The **`[T]`** or Top modifier selects bits 31 to 16 as those specified by #I16. Bits 15 to 0 set to zero.

The **`[MT]`** or Mask Top modifier is similar to the T case except bits 15 to 0 are set to one.

The **`[MB]`** or Mask Bottom modifier sets bits 15 to 0 from #I16 with bits 31 to 16 set to one.

2.2.42. TST Dx.r,Rx.r

Updates all the condition flags to indicate the result of an equivalent bit-wise AND operation based on the two input parameters.

See encoding diagram 3.2.42

Syntax

TST	**Src1,Src2**
Src1	Dx.r
Src2	Dx.r | Ax.r | RD

Operation

```
-1:       O2RSrc   = O2R(Src),
   [L] -> O2RSrc.o = O2R(Src.o);
0:        DUAnd(Src1, Src2),
          'Flags   = DUFlags,
   [L] -> DUAnd(Src1.o, Src2.o);
```

Examples

```
  TST       D0.7,A1.7
  TST       D0Re0,D1.7                                          ;
  TST       D0Re0,RD
```

Usage notes

See section 2.1.8 Condition-flag setting operations.

See Operand 2 replace in section 3.1.3

2.2.43. XOR De.r,Dx.r,De.r

Performs a bitwise XOR of two operands and stores the result in the source data unit.
See encoding diagram 3.2.43

Syntax

XOR[S] **Dst,Src1,Src2**

Dst	De.r
Src1	Dx.r
Src2	De.r

Operation

```
0:          'Dst   = DUXor(Src1, Src2),
   [S] ->   'Flags = DUFlags;
```

Examples

```
  XOR       D0.7,D0.7,D0.7
  XORS      D1Re0,D1.7,D1.7
```

Usage notes

2.2.44. XOR De.e,Dx.r,#I16

Perform a bitwise XOR with 16-bit immediate data in the data units using a the same register for both source or destination.

See encoding diagram 3.2.44

Syntax

XOR[S][T|MT|MB] **Dst,Src1,Src2**

Dst	De.r
Src1	Dx.r
Src2	#I16

Operation

```
0:          IMMSrc2 = Src2,
    [T]  -> IMMSrc2 <<= 16,
    [MT] -> IMMSrc2 |= 0x0000FFFF,
    [MB] -> IMMSrc2 |= 0xFFFF0000,
            'Dst    = DUXor(Src1, IMMSrc2),
    [S]  -> 'Flags  = DUFlags;
```

Examples

```
  XOR      D0Re0,D0Re0,#-0x8000          ; XOR with 0xFFFF8000
  XORT     D1Re0,D1Re0,#0x7fff           ; XOR with 0x7fff0000
  XORMT    D0.7,D0.7,#HI(0x7fff1234)     ; XOR with 0x7fffffff ...
  XORMB    D0.7,D0.7,#LO(0x7fff1234)     ;  ... then 0xffff1234
```

Usage notes

The **`[T]`** or Top modifier selects bits 31 to 16 as those specified by #I16. Bits 15 to 0 set to zero.

The **`[MT]`** or Mask Top modifier is similar to the T case except bits 15 to 0 are set to one.

The **`[MB]`** or Mask Bottom modifier sets bits 15 to 0 from #I16 with bits 31 to 16 set to one.

2.3. Extended general purpose instructions

2.3.1. ADD Ae.r,Ax.r,Rx.r

Performs a cross-unit arithmetic addition into an address unit destination.
See encoding diagram 3.3.1

Syntax

ADD **Dst,Src1,Src2**

Dst	Ae.r
Src1	Ax.r | CPCx
Src2	Ax.r | Dx.r | RD

Operation

```
-1:        O2RSrc2 = O2R(Src2);
0 :        'Dst    = AUAdd(Src1, O2RSrc2);
```

Examples

```
 ADD       A0FrP,CPC0,RD
 ADD       A1.7,A1.7,D0.7
 ADD       A1.7,A1LbP,A0.7
```

Usage notes

See Operand 2 replace in section 3.1.3

2.3.2. ADDcc Rx.r,Ax.r,Ae.r

Performs a conditional address unit arithmetic addition into any destination.
See encoding diagram 3.3.2

Syntax

ADDcc **Dst,Src1,Src2**

Dst	Ax.r | Dx.r | PC.r | RAxx | CT.r | TR.r | TT.r
Src1	Ax.r | CPCx
Src2	Ae.r | CPCe

Operation

```
0:           ADDVal = AUAdd(Src1, Src2);
1: <cc> ->   'Dst   = ADDVal;
```

Examples

```
 ADDMI     PCX,CPC1,CPC1
 ADDCS     TXRPT,A0FrP,A0.7
 ADDEQ     RAMX,A1LbP,CPC1.
```

Usage notes

See section 2.1.9 Conditional operations.

2.3.3. ADDcc Rx.r,Ax.r,Rx.r

Performs a conditional cross-unit arithmetic addition with an address unit as the primary unit.

See encoding diagram 3.3.3

Syntax

ADDcc **Dst,Src1,Src2**

Dst	Ax.r | Dx.r | PC.r | RAxx | CT.r | TR.r | TT.r
Src1	Ax.r | CPCx
Src2	Ax.r | Dx.r | RD

Operation

```
-1:           O2RSrc2 = O2R(Src2);
0:            ADDVal  = AUAdd(Src1, O2RSrc2);
1: <cc> ->    'Dst    = ADDVal;
```

Examples

```
ADDCS     TXMASKI,CPC1,A0FrP
ADDVC     RAMX,A0FrP,A0FrP
ADDHI     D1.7,A0.7,D1.7
```

Usage notes

See section 2.1.9 Conditional operations.

See Operand 2 replace in section 3.1.3

2.3.4. ADD Rx.r,Ax.r,#X8

Performs a cross-unit arithmetic addition with 8-bit immediate data and an address unit as the primary unit.

See encoding diagram 3.3.4

Syntax

ADD **Dst,Src1,Src2**

Dst	Ax.r | Dx.r | PC.r | RAxx | CT.r | TR.r | TT.r
Src1	Ax.r | CPCx
Src2	#X8

Operation

```
0:          ADDVal = AUAdd(Src1, Src2);
1:          'Dst   = ADDVal;
```

Examples

```
 ADD        A0.7,A1.7,#0x7f
 ADD        D0.7,A0.7,#0x7f
 ADD        TXMASKI,A1.7,#0x7f
```

Usage notes

2.3.5. ADDcc Ae.r,Ax.r,#X8

Performs a conditional address unit arithmetic addition with 8-bit immediate data.

See encoding diagram 3.3.5

Syntax

ADDcc **Dst,Src1,Src2**

Dst	Ae.r
Src1	Ax.r | CPCx
Src2	#X8

Operation

```
0:           ADDVal = AUAdd(Src1, Src2);
1: <cc> ->  'Dst    = ADDVal;
```

Examples

```
ADDCS     A1.7,A1.7,#0x80
ADDLE     A0.7,CPC0,#0x80
```

Usage notes

See section 2.1.9 Conditional operations.

2.3.6. ADD De.r,Dx.r,Rx.r

Performs a cross-unit arithmetic addition into a data unit destination.

See encoding diagram 3.3.6

Syntax

ADD[S] **Dst,Src1,Src2**

Dst	De.r
Src1	Dx.r
Src2	Dx.r | Ax.r | RD

Operation

```
-1:         O2RSrc2 = O2R(Src2);
0:          'Dst    = DUAdd(Src1, O2RSrc2),
    [S] -> 'Flags  = DUFlags;
```

Examples

```
 ADD       D0.7,D0.7,A0.7
 ADD       D0.7,D0.7,RD
 ADDS      D0.7,D0.7,A1.7
```

Usage notes

See Operand 2 replace in section 3.1.3

2.3.7. ADDcc Rx.r,Dx.r,De.r

Performs a conditional data unit arithmetic addition into any destination.

See encoding diagram 3.3.7

Syntax

ADD[S]cc **Dst,Src1,Src2**

Dst	Ax.r | Dx.r | PC.r | RAxx | CT.r | TR.r | TT.r
Src1	Dx.r
Src2	De.r

Operation

```
0:                    ADDVal = DUAdd(Src1, Src2),
   ([S] && <cc>) -> 'Flags = DUFlags;
1:           <cc> -> 'Dst   = ADDVal;
```

Examples

```
 ADDCS     A1.7,D0.7,D0Re0
 ADDSMI    RAWX,D0Re0,D0.7
 ADDNE     TTEXEC,D1Re0,D1.7
```

Usage notes

See section 2.1.9 Conditional operations.

2.3.8. ADDcc Rx.r,Dx.r,Rx.r

Performs a conditional cross-unit arithmetic addition with a data unit as the primary unit.
See encoding diagram 3.3.8

Syntax

ADD[S]cc Dst,Src1,Src2

Dst	Ax.r | Dx.r | PC.r | RAxx | CT.r | TR.r | TT.r
Src1	Dx.r
Src2	Ax.r | Dx.r | RD

Operation

```
-1:                 O2RSrc2 = O2R(Src2);
0:                  ADDVal  = DUAdd(Src1, O2RSrc2),
   ([S] && <cc>) -> 'Flags  = DUFlags;
1:          <cc> -> 'Dst    = ADDVal;
```

Examples

```
ADDGE    A0.7,D0.7,RD
ADDSGT   RAWX,D0Re0,A0.7
ADDNV    TXRPT,D1Re0,D0.7
```

Usage notes

See section 2.1.9 Conditional operations.
See Operand 2 replace in section 3.1.3

2.3.9. ADD Rx.r,Dx.r,#X8

Performs a cross-unit arithmetic addition with 8-bit immediate data and a data unit as the primary unit. See encoding diagram 3.3.9

Syntax

ADD[S] **Dst,Src1,Src2**

Dst	Ax.r | Dx.r | PC.r | RAxx | CT.r | TR.r | TT.r
Src1	Dx.r
Src2	#X8

Operation

```
0:         ADDVal = DUAdd(Src1, Src2);
    [S] -> 'Flags = DUFlags;
1:         'Dst   = ADDVal;
```

Examples

```
 ADD       A0.7,D0.7,#0x7f
 ADD       D0.7,D1Re0,#0xff
 ADDS      TXMASKI,D1.7,#0x7f
```

Usage notes

2.3.10. ADDcc De.r,Dx.r,#X8

Performs a conditional data unit arithmetic addition with 8-bit immediate data.
See encoding diagram 3.3.10

Syntax

ADD[S]cc **Dst,Src1,Src2**

Dst	De.r
Src1	Dx.r
Src2	#X8

Operation

```
0:          <cc>  -> 'Dst   = DUAdd(Src1, Src2),
   ([S] && <cc>) -> 'Flags = DUFlags;
```

Examples

```
 ADDCS     D1.7,D1.7,#0x80
 ADDSNV    D0.7,D0.7,#0xff
```

Usage notes

See section 2.1.9 Conditional operations.

2.3.11. AND De.r,Dx.r,Rx.r

Performs a cross-unit bitwise AND into a data unit destination.

See encoding diagram 3.3.11

Syntax

AND[S] **Dst,Src1,Src2**

Dst	De.r
Src1	Dx.r
Src2	Dx.r | Ax.r | RD

Operation

```
-1:        O2RSrc2 = O2R(Src2);
0 :        'Dst    = DUAnd(Src1, O2RSrc2),
    [S] -> 'Flags  = DUFlags;
```

Examples

```
  AND       D0.7,D0.7,A0.7
  AND       D0Re0,D0Re0,D1.7
  ANDS      D1.7,D1Re0,RD
```

Usage notes

See Operand 2 replace in section 3.1.3

2.3.12. ANDcc Rx.r,Dx.r,De.r

Performs a conditional data unit bitwise AND into any destination.

See encoding diagram 3.3.12

Syntax

AND[S]cc **Dst,Src1,Src2**

Dst	Ax.r | Dx.r | PC.r | RAxx | CT.r | TR.r | TT.r
Src1	Dx.r
Src2	De.r

Operation

```
0:          <cc>  -> 'Dst   = DUAnd(Src1, Src2),
   ([S] && <cc>) -> 'Flags = DUFlags;
```

Examples

```
  ANDNE     A0.7,D1.7,D1.7
  ANDCS     RAMX,D1Re0,D1Re0
  ANDSEQ    PC,D1.7,D1Re0
```

Usage notes

See section 2.1.9 Conditional operations.

2.3.13. ANDcc Rx.r,Dx.r,Rx.r

Performs a conditional cross-unit bitwise AND with a data unit as the primary unit.

See encoding diagram 3.3.13

Syntax

AND[S]cc **Dst,Src1,Src2**

Dst	Ax.r | Dx.r | PC.r | RAxx | CT.r | TR.r | TT.r
Src1	Dx.r
Src2	Dx.r | Ax.r | RD

Operation

```
-1:                  O2RSrc2 = O2R(Src2);
0 :                  ANDVal  = DUAnd(Src1, O2RSrc2),
    ([S] && <cc>) -> 'Flags  = DUFlags;
1:           <cc> -> 'Dst    = ANDVal;
```

Examples

```
  ANDGE     GTEXEC,D0Re0,A0.7
  ANDGT     RA,D1.7,RD
  ANDSGT    TXMASKI,D0Re0,A0.7
  ANDSVS    D1.7,D1Re0,A1.7
```

Usage notes

See section 2.1.9 Conditional operations.

See Operand 2 replace in section 3.1.3

2.3.14. AND Rx.r,Dx.r,#X8

Performs a cross-unit bitwise AND with 8-bit immediate data and a data unit as the primary unit.
See encoding diagram 3.3.14

Syntax

AND[S] **Dst,Src1,Src2**

Dst	Ax.r | Dx.r | PC.r | RAxx | CT.r | TR.r | TT.r
Src1	Dx.r
Src2	#X8

Operation

```
0:          ANDVal = DUAnd(Src1, Src2),
    [S] -> 'Flags = DUFlags;
1:          'Dst   = ANDVal;
```

Examples

```
 AND        A0.7,D0.7,#0x7f
 AND        D0.7,D1Re0,#0xff
 ANDS       TXMASKI,D1.7,#0x7f
```

Usage notes

2.3.15. ANDcc De.r,Dx.r,#X8

Performs a conditional data unit bitwise AND with 8-bit immediate data.

See encoding diagram 3.3.15

Syntax

AND[S]cc **Dst,Src1,Src2**

Dst	De.r
Src1	Dx.r
Src2	#X8

Operation

```
0:         <cc>  -> 'Dst   = DUAdd(Src1, Src2),
   ([S] && <cc>) -> 'Flags = DUFlags;
```

Examples

```
ANDGT     D1.7,D1.7,#0xff
ANDSNE    D0.7,D0.7,#0x80
```

Usage notes

See section 2.1.9 Conditional operations.

2.3.16. CMPcc Dx.r,De.r

Performs a conditional comparison and updates all the condition flags to indicate the result of an equivalent subtract operation based on the two input parameters.

See encoding diagram 3.3.16

Syntax

CMPcc **Src1,Src2**

Src1	Dx.r
Src2	De.r

Operation

```
0:           DUSub(Src1, Src2),
   <cc> ->   'Flags = DUFlags;
```

Examples

```
  CMPCS     D0Re0,D0Re0
  CMPMI     D1Re0,D1.7
```

Usage notes

See section 2.1.8 Condition-flag setting operations.

2.3.17. CMPcc Dx.r,Rx.r

Performs a conditional comparison and updates all the condition flags to indicate the result of an equivalent subtract operation based on the two input parameters.

See encoding diagram 3.3.17

Syntax

CMPcc **Src1,Src2**

Src1	Dx.r
Src2	Dx.r | Ax.r | RD

Operation

```
-1:          O2RSrc2 = O2R(Src2);
0:           DUSub(Src1, O2RSrc2),
   <cc> ->   'Flags = DUFlags;
```

Examples

```
  CMPGT      D0.7,A1.7
  CMPNV      D0Re0,D1.7
  CMPVS      D0Re0,RD
```

Usage notes

See section 2.1.8 Condition-flag setting operations.

See Operand 2 replace in section 3.1.3

2.3.18. CMPcc Dx.r,#X8

Performs a conditional comparison and updates all the condition flags to indicate the result of an equivalent subtract operation based on the two input parameters.

See encoding diagram 3.3.18

Syntax

CMPcc **Src1,Src2**

Src1	Dx.r
Src2	#X8

Operation

```
0:          DUSub(Src1, Src2),
   <cc> ->  'Flags = DUFlags;
```

Examples

```
  CMPLE     D0.7,#0x7f
  CMPGE     D1Re0,#0x7f
```

Usage notes

See section 2.1.8 Condition-flag setting operations.

2.3.19. CPRx [BBxUaRoPp|BBxUaS6Pp],CP.n

Stores data from a single co-processor read to memory using full address mode support and all sizes up to 64-bit.

See encoding diagram 3.3.19

Syntax

CPR[B|W|D|L] **[Dst],Src**

[Dst]	[BBxUaRoPp|BBxUaS6Pp]
Src	CP.n

Operation

```
0:          Addr   = BUAddr(Dst),
            Size   = [B|W|D|L],
            Data   = Src;
2:          MemStore(Addr, Size, Data);
```

Examples

```
  CPRB      [A0StP+#-0x1], CP.0
  CPRW      [D0Re0+#-0x2], CP.5
  CPRD      [D1.7++D1Re0], CP.4
  CPRL      [A1LbP+#0x0], CP.2
```

Usage notes

See section 2.1.4 Address modes.

2.3.20. GETL BBx.r,BBo.e,[BBxUaRoPp|BBxUaS6Pp]

Loads a pair of registers from memory with full address mode support.

See encoding diagram 3.3.20

Syntax

GETL **Dst1,Dst2,[Src]**

Dst1	BBx.r
Dst2	BBo.e
[Src]	[BBxUaRoPp|BBxUaS6Pp]

Operation

```
0:          Addr         = BUAddr(Src),
            Size         = L;
2:          Data.o, Data = MemLoad(Addr, Size);
3:          'Dst1        = Data;
            'Dst2        = Data.o;
```

Examples

```
  GETL      A0.7,A1.7,[D1Re0--]
  GETL      D0Re0,D1Re0,[D0.7+#-0x100++]
  GETL      A1.7,A0.7,[D1Re0+#-0x100++]
  GETL      A1.7,A0.7,[D1Re0+D1.7]
  GETL      D0.7,D1.7,[A1LbP+A1.7]
```

Usage notes

The same address/data unit may be used to generate the memory address and specify the register to be loaded.

See section 2.1.4 Address modes.

2.3.21. GETx Ux.r,[BBxUaRoPp|BBxUaS6Pp]

Loads a single register from memory with a full set of addressing modes and sizes up to 32-bit.

See encoding diagram 3.3.21

Syntax

GET[B|W|D] **Dst,[Src]**

Dst	Ax.r | Dx.r | PC.r | CT.r | TR.r
[Src]	[BBxUaRoPp|BBxUaS6Pp]

Operation

```
0:          Addr         = BUAddr(Src),
            Size         = [B|W|D];
2:          Data.o, Data = MemLoad(Addr, Size);
3:          'Dst1        = Data;
```

Examples

```
  GETB      A0.7,[D1Re0--]
  GETD      D0Re0,[D0.7+#-0x80++]
  GETW      A1.7,[D1Re0+#-0x40++]
  GETD      A1.7, [D1Re0+D1.7]
  GETB      D0.7,[A1LbP+A1.7]
```

Usage notes

The same address/data unit may be used to generate the memory address and specify the register to be loaded.

PC.r | CT.r | TR.r | TT.r can only be loaded from a 32-bit **`[D]`** instruction.

See section 2.1.4 Address modes.

2.3.22. MDRD RD..RD..

Flushes a group of up to eight read data words from the read pipeline.

See encoding diagram 3.3.22

Syntax

MDRD RD, RD, RD, RD, RD, RD, RD, RD

Operation

```
n-1:        RDData = XUMov(RD);
n:          Discard(RDData);
```

Examples

```
  MDRD      #0x5
  MDRD      #0x6
  MDRD      #0x7
  MDRD      #0x8
```

Usage notes

If the read pipeline does not contain enough data to satisfy the needs of this instruction an exception will be caused or the thread will be halted.

2.3.23. MMOVx BBx.r..BBe.e..,RD

Allows up to eight read data words to be drained from the read pipeline to one or two units.

See encoding diagram 3.3.23

Syntax

MMOV[D|L] **Dst1,..,Dst8,Src**

Dst	BBx.r…BBe.r	(*restricted to Dx.(0-15) and Ax.(0-7))
Src	RD	

Operation

```
n-1:  RDData, RDData.o = XUMov(RD);
n+0:            'Dst   = RDData,       (bottom 32 bits of RD data entry)
         [L] -> 'Dst.o = RDData.o;     (top 32 bits of RD data entry)
```

Examples

```
  MMOVD     A0.3,A0.7,RD
  MMOVD     A0FrP,A0.2,A0.3,A0.7,RD
  MMOVL     D0Re0,D0Ar6,D0.7,RD
  MMOVL     D0Re0,D0Ar6,D0Ar4,D0.7,RD
```

Usage notes

*Global (Data and Address) registers are not accessible by these instructions.

The Dst registers specified must be in the same unit.

When **`[L]`** is specified a dual unit operation is performed using the same register numbers in each unit (with D0/D1 and A0/A1 being the allowed pairs). In this case, the low 32-bits of the data come from the unit specified, and the remaining 32-bits come from the implied unit (i.e. A0 implies A1, A1 implies A0, D0 implies D1 and D1 implies D0).

If the read pipeline does not contain enough data to satisfy the needs of this instruction an exception will be caused or the thread will be halted.

2.3.24. MMOVx RAxx..RAee..,[BBx.r]

Issues a group of up to eight read addresses to the read pipeline.

See encoding diagram 3.3.24

Syntax

MMOV[D|L] **Dst1,..,Dst8,Src**

Dst	RAxx,RAee…
Src	[BBx.r++]

Operation

```
n+0:         Addr         = BUAddr(Src),
             Size         = [D|L];
n+2:  [D] -> Data         = RAMov(Addr, Size),
      [L] -> Data, Data.o = RAMov(Addr, Size) ;
n+3:         'RD          = RAMod(Data, Src, Size);
```

Examples

```
MMOVD     RA,RA,RA,RA,RA,[A1LbP++]
MMOVD     RA,RA,RA,RA,RA,RA,RA,RA,[D0.7++]
MMOVL     RA,RA,RA,RA,RA,[A0FrP++]
MMOVL     RAWX,RAWX,RAWX,RAWX,[D0.7++]
```

Usage notes

Dst read pipeline address ports must be of the same type.

[Src] only uses post-increment addressing. The address issued increments by either 4 or 8 depending upon whether `[D]` or `[L]` is specified.

2.3.25. MOV Ae.r,Rx.r

Load an address unit register from a value taken from another unit or the RD port.

See encoding diagram 3.3.25

Syntax

MOV	**Dst,Src**
Dst	Ax.r
Src	Ao.r | Dx.r | RD

Operation

```
-1:         O2RData = O2R(Src);
0:          'Dst    = O2RData;
```

Examples

```
  MOV       A0.3,A1.2
  MOV       A0.3,RD
  MOV       A1.3,D0.2
  MOV       A1LbP,RD
```

Usage notes

The data transferred by this operation is only up to 32-bits as for other ALU operations. For a 64-bit MOV from RD see section 2.6.22.

See Operand 2 replace in section 3.1.3

2.3.26. MOV Dx.r,Rx.r

Load a data unit register with a value taken from another unit or the RD port.

See encoding diagram 3.3.26

Syntax

MOV[S]	**Dst,Src**
Dst	Dx.r
Src	Do.r | Ax.r | RD

Operation

```
-1:         O2RData = O2R(Src);
0:          'Dst    = O2RData,
     [S] -> 'Flags  = DUFlags;
```

Examples

```
  MOVS      D0.7,A1.3
  MOV       D0.7,RD
  MOV       D1.7,D0Ar6
  MOVS      D1Re0,RD
```

Usage notes

The data transferred by this operation is only up to 32-bits as for other ALU operations. For a 64-bit MOV from RD see section 2.6.22.

See Operand 2 replace in section 3.1.3

2.3.27. MOVx RAxx,[BBxUaRoPp|BBxUaS6Pp]

Issues a read address to the read pipeline.

See encoding diagram 3.3.27

Syntax

MOV[B|W|D|L] **Dst,[Src]**

Dst	RAxx
[Src]	[BBxUaRoPp | BBxUaS6Pp]

Operation

```
0:          Addr          = BUAddr(Src),
            Size          = [B|W|D|L];
2: [!L] -> Data          = RAMov(Addr, Size),
    [L] -> Data, Data.o = RAMov(Addr, Size) ;
3:          'RD           = RAMod(Data, Src, Size);
```

Examples

```
  MOVB      RABZ,[A1LbP++#0x1f]
  MOVD      RAM8X,[A0.7+#-0x80++]
  MOVL      RAM8X,[A1LbP--]
  MOVW      RAMX,[D1.7++#0x3e]

  MOVB      RAM8X,[A1.7+A1.7++]
  MOVD      RAMX,[A1.7+A1LbP]
  MOVW      RAWX,[D1.7++D1Re0]
  MOVL      RABZ,[D1.7+D1.7]
```

Usage notes

See section 2.1.4 Address modes.

2.3.28. MUL De.r,Dx.r,Rx.r

Performs a cross-unit unit multiply and stores the result in a data unit register.

See encoding diagram 3.3.28

Syntax

MUL[D|W] **Dst,Src1,Src2**

Dst	De.r
Src1	Dx.r
Src1	Do.r | Ax.r | RD

Operation

```
-1:        O2RSrc2 = O2R(Src2);
0:         MULVal  = DUMul(Src1, O2RSrc2),
3:         'Dst    = MULVal;
```

Examples

```
  MULD      D0.7,D0.7,A0.7
  MULD      D0.7,D0.7,D1.7
  MULW      D1.7,D1.7,A0.7
  MULW      D1.7,D1.7,RD
```

Usage notes

MULW performs an unsigned 16 x 16 multiply which takes multiple cycles. The returned value is the full 32-bit result.

MULD is an unsigned 32 x 32 multiply which takes multiple cycles. The returned value is the lower 32-bits of the 64-bit result.

See Operand 2 replace in section 3.1.3

2.3.29. MUL BBx.r,Dx.r,De.r

Performs a cross-unit unit multiply with a data unit as the primary unit.

See encoding diagram 3.3.29

Syntax

MUL[D|W] **Dst,Src1,Src2**

Dst	BBx.r
Src1	Dx.r
Src1	De.r

Operation

```
0:          MULVal  = DUMul(Src1, Src2),
3:          'Dst    = MULVal;
```

Examples

```
  MULD      A0.7,D0.7,D0.7
  MULD      A1.7,D0.7,D0Re0
  MULW      D0.7,D1.7,D1.7
  MULW      D1.7,D0.7,D0.7
```

Usage notes

MULW performs an unsigned 16 x 16 multiply which takes multiple cycles. The returned value is the full 32-bit result.

MULD is an unsigned 32 x 32 multiply which takes multiple cycles. The returned value is the lower 32-bits of the 64-bit result.

2.3.30. MUL BBx.r,Dx.r,Rx.r

Performs a cross-unit unit multiply with a data unit as the primary unit.

See encoding diagram 3.3.30

Syntax

MUL[D|W] **Dst,Src1,Src2**

Dst	BBx.r
Src1	Dx.r
Src1	Do.r | Ax.r | RD

Operation

```
-1:        O2RSrc2 = O2R(Src2);
0:         MULVal  = DUMul(Src1, O2RSrc2),
3:         'Dst    = MULVal;
```

Examples

```
  MULD      A0.7,D0.7,A0.7
  MULD      A0.7,D1.7,D0.7
  MULD      D0.7,D1Re0,RD
  MULW      D1.7,D0.7,D1.7
  MULW      D1Re0,D0.7,RD
```

Usage notes

MULW performs an unsigned 16 x 16 multiply which takes multiple cycles. The returned value is the full 32-bit result.

MULD is an unsigned 32 x 32 multiply which takes multiple cycles. The returned value is the lower 32-bits of the 64-bit result.

See Operand 2 replace in section 3.1.3

2.3.31. MUL BBx.r,Dx.r,#X8

Performs a cross-unit multiply with 8-bit immediate data.

See encoding diagram 3.3.31

Syntax

MUL[D|W][T] **Dst,Src1,Src2**

Dst	BBx.r
Src1	Dx.r
Src1	#X8

Operation

```
0:          MULVal  = DUMul(Src1, Src2),
3:          'Dst    = MULVal;
```

Examples

```
  MULD      A0.7,D0.7,#0x7f
  MULD      D0Re0,D1.7,#0x80
  MULW      A0.7,D1Re0,#0x80
  MULW      D1.7,D1Re0,#0x7f
```

Usage notes

MULW performs an unsigned 16 x 16 multiply which takes multiple cycles. The returned value is the full 32-bit result.

MULD is an unsigned 32 x 32 multiply which takes multiple cycles. The returned value is the lower 32-bits of the 64-bit result.

2.3.32. MULcc De.r,Dx.r,De.r

Performs a conditional internal data unit multiply.

See encoding diagram 3.3.32

Syntax

MUL[D|W]cc **Dst,Src1,Src2**

Dst	De.r
Src1	Dx.r
Src1	De.r

Operation

```
0:          MULVal  = DUMul(Src1, Src2),
2: <cc> -> 'Dst     = MULVal;
```

Examples

```
  MULDEQ    D0.7,D0.7,D0.7
  MULDMI    D0.7,D0.7,D0Re0
  MULWCS    D1.7,D1Re0,D1.7
  MULWNE    D1.7,D1.7,D1.7
```

Usage notes

MULW performs an unsigned 16 x 16 multiply which takes multiple cycles. The returned value is the full 32-bit result.

MULD is an unsigned 32 x 32 multiply which takes multiple cycles. The returned value is the lower 32-bits of the 64-bit result .

See section 2.1.9 Conditional operations.

2.3.33. MULcc De.r,Dx.r,Rx.r

Performs a conditional cross-unit unit multiply and stores the result in a data unit register.

See encoding diagram 3.3.33

Syntax

MUL[D|W]cc **Dst,Src1,Src2**

Dst	De.r
Src1	Dx.r
Src1	Do.r | Ax.r | RD

Operation

```
-1:         O2RSrc2 = O2R(Src2);
0:          MULVal  = DUMul(Src1, O2RSrc2),
3: <cc> -> 'Dst     = MULVal;
```

Examples

```
  MULDGE    D0.7,D0.7,A0.7
  MULDGE    D0.7,D0.7,RD
  MULWNV    D1Re0,D1Re0,A0.7
  MULWNV    D1.7,D1Re0,RD
```

Usage notes

MULW performs an unsigned 16 x 16 multiply which takes multiple cycles. The returned value is the full 32-bit result.

MULD is an unsigned 32 x 32 multiply which takes multiple cycles. The returned value is the lower 32-bits of the 64-bit result .

See section 2.1.9 Conditional operations.

See Operand 2 replace in section 3.1.3

2.3.34. MULcc De.r,Dx.r,#X8

Performs a conditional data unit multiply with 8-bit immediate data.

See encoding diagram 3.3.34

Syntax

MUL[D|W]cc **Dst,Src1,Src2**

Dst	De.r
Src1	Dx.r
Src1	#X8

Operation

```
0:          MULVal  = DUMul(Src1, Src2),
2: <cc> ->     'Dst = MULVal;
```

Examples

```
  MULDGE    D0.7,D0.7,#0x80
  MULDGT    D1Re0,D1Re0,#0x80
  MULWGE    D1.7,D1Re0,#0x7f
```

Usage notes

MULW performs an unsigned 16 x 16 multiply which takes multiple cycles. The returned value is the full 32-bit result.

MULD is an unsigned 32 x 32 multiply which takes multiple cycles. The returned value is the lower 32-bits of the 64-bit result .

See section 2.1.9 Conditional operations.

2.3.35. NEG Ae.r,Rx.r

Load an address unit register with negated data taken from another unit or the RD port.
See encoding diagram 3.3.35

Syntax

NEG	**Dst,Src**
Dst	Ax.r
Src	Ao.r | Dx.r | RD

Operation

```
-1:         O2RData = O2R(Src);
0:          Zero    = 0,
            'Dst    = AUSub(Zero, O2RData);
```

Examples

```
  NEG       A0.3,A1.2
  NEG       A0.3,RD
  NEG       A1.3,D0.2
  NEG       A1LbP,RD
```

Usage notes

The data transferred by this operation is only up to 32-bits as for other ALU operations.
See Operand 2 replace in section 3.1.3

2.3.36. NEG Dx.r,Rx.r

Load a data unit register with negated data from another unit or the RD port.

See encoding diagram 3.3.36

Syntax

NEG[S] **Dst,Src**

Dst	Dx.r
Src	Do.r | Ax.r | RD

Operation

```
-1:          O2RData = O2R(Src);
0:           Zero    = 0,
             'Dst    = DUSub(Zero, O2RData),
     [S] -> 'Flags   = DUFlags;
```

Examples

```
  NEGS      D0.7,A1.3
  NEG       D0.7,RD
  NEG       D1.7,D0Ar6
  NEGS      D1Re0,RD
```

Usage notes

The data transferred by this operation is only up to 32-bits as for other ALU operations. For a 64-bit MOV from RD see section 2.6.22.

See Operand 2 replace in section 3.1.3

2.3.37. OR De.r,Dx.r,Rx.r

Performs a cross-unit bitwise OR into a data unit destination.

See encoding diagram 3.3.37

Syntax

OR[S] **Dst,Src1,Src2**

Dst	De.r
Src1	Dx.r
Src2	Dx.r | Ax.r | RD

Operation

```
-1:         O2RSrc2 = O2R(Src2);
0 :         'Dst    = DUOr(Src1, O2RSrc2),
    [S] -> 'Flags   = DUFlags;
```

Examples

```
  OR       D0.7,D0.7,A0.7
  OR       D0Re0,D0Re0,D1.7
  ORS      D1.7,D1Re0,RD
```

Usage notes

See Operand 2 replace in section 3.1.3

2.3.38. ORcc Rx.r,Dx.r,De.r

Performs a conditional data unit bitwise OR into any destination.
See encoding diagram 3.3.38

Syntax

OR[S]cc **Dst,Src1,Src2**

Dst	Ax.r | Dx.r | PC.r | RAxx | CT.r | TR.r | TT.r
Src1	Dx.r
Src2	De.r

Operation

```
0:         <cc>  -> 'Dst   = DUOr(Src1, Src2),
   ([S] && <cc>) -> 'Flags = DUFlags;
```

Examples

```
  ORNE     A0.7,D1.7,D1.7
  ORCS     RAMX,D1Re0,D1Re0
  ORSEQ    PC,D1.7,D1Re0
```

Usage notes

See section 2.1.9 Conditional operations.

2.3.39. ORcc Rx.r,Dx.r,Rx.r

Performs a conditional cross-unit bitwise OR with a data unit as the primary unit.

See encoding diagram 3.3.39

Syntax

OR[S]cc **Dst,Src1,Src2**

Dst	Ax.r | Dx.r | PC.r | RAxx | CT.r | TR.r | TT.r
Src1	Dx.r
Src2	Dx.r | Ax.r | RD

Operation

```
-1:                 O2RSrc2 = O2R(Src2);
0 :                 ORVal  = DUOr(Src1, O2RSrc2),
    ([S] && <cc>) -> 'Flags  = DUFlags;
1:           <cc> -> 'Dst    = ORVal;
```

Examples

```
  ORGE     GTEXEC,D0Re0,A0.7
  ORGT     RA,D1.7,RD
  ORSGT    TXMASKI,D0Re0,A0.7
  ORSVS    D1.7,D1Re0,A1.7
```

Usage notes

See section 2.1.9 Conditional operations.

See Operand 2 replace in section 3.1.3

2.3.40. OR Rx.r,Dx.r,#X8

Performs a cross-unit bitwise OR with 8-bit immediate data and a data unit as the primary unit.
See encoding diagram 3.3.40

Syntax

OR[S] **Dst,Src1,Src2**

Dst	Ax.r | Dx.r | PC.r | RAxx | CT.r | TR.r | TT.r
Src1	Dx.r
Src2	#X8

Operation

```
0:          ORVal = DUOr(Src1, Src2),
    [S] -> 'Flags = DUFlags;
1:          'Dst   = ORVal;
```

Examples

```
 OR       A0.7,D0.7,#0x7f
 OR       D0.7,D1Re0,#0xff
 ORS      TXMASKI,D1.7,#0x7f
```

Usage notes

2.3.41. ORcc De.r,Dx.r,#X8

Performs a conditional data unit bitwise OR with 8-bit immediate data.

See encoding diagram 3.3.41

Syntax

OR[S]cc **Dst,Src1,Src2**

Dst	De.r
Src1	Dx.r
Src2	#X8

Operation

```
0:          <cc>  -> 'Dst   = DUAdd(Src1, Src2),
   ([S] && <cc>) -> 'Flags = DUFlags;
```

Examples

```
ORGT     D1.7,D1.7,#0xff
ORSNE    D0.7,D0.7,#0x80
```

Usage notes

See section 2.1.9 Conditional operations.

2.3.42. SETL [BBxUaRoPp|BBxUaS6Pp],BBx.r,BBo.e

Stores a pair of registers to memory using full address mode support.

See encoding diagram 3.3.42

Syntax

SETL **[Dst],Src1,Src2**

[Dst]	[BBxUaS6Pp]	[AxUaRoPp|	[DxUaRoPp|
Src1	BBx.r	Dx.r	Ax.r
Src2	BBo.e	Do.e	Ao.e

Operation

```
0:              Addr   = BUAddr(Dst),
                Size   = L,
                Data   = Src1, Src2;
2:              MemStore(Addr, Size, Data);
```

Examples

```
  SETL      [A0.3+#-0x100],D1Re0,D0Re0
  SETL      [A0FrP+#0xF8],A1LbP,A0FrP
  SETL      [D0Re0+#0x10++],D1Re0,D0Re0
  SETL      [D1Re0--],A1LbP,A0FrP
```

Usage notes

The same address/data unit may be used to generate the memory address and specify the register to be stored provided the addressing mode used does not attempt to use a register offset; only immediate/zero offsets are supported in these cases.

See section 2.1.4 Address modes.

2.3.43. SETx [BBxUaRoPp|BBxUaS6Pp],Ux.r

Stores a single register to memory using full address mode support and all sizes up to 32-bit.

See encoding diagram 3.3.43

Syntax

SET[B|W|D] **[Dst],Src**

[Dst]	[BBxUaRoPp|BBxUaS6Pp]
Src	Ax.r | Dx.r | PC.r | CT.r | TR.r

Operation

```
0:          Addr   = BUAddr(Dst),
            Size   = [B|W|D],
            Data   = Src;
2:          MemStore(Addr, Size, Data);
```

Examples

```
  SETB      [A0FrP+#0x7ff],A0.7
  SETD      [D0Re0+#-0x2000],A1.7
  SETW      [A1LbP+#0xffe],D1Re0
  SETW      [D0Re0+#0xffe],D1.7
```

Usage notes

The same address/data unit may be used to generate the memory address and specify the register to be loaded.

See section 2.1.4 Address modes.

2.3.44. SUB Ae.r,Ax.r,Rx.r

Performs a cross-unit arithmetic subtraction into an address unit destination.

See encoding diagram 3.3.44

Syntax

SUB **Dst,Src1,Src2**

Dst	Ae.r
Src1	Ax.r | CPCx
Src2	Ax.r | Dx.r | RD

Operation

```
-1:         O2RSrc2 = O2R(Src2);
0 :         'Dst    = AUSub(Src1, O2RSrc2);
```

Examples

```
SUB       A0FrP,CPC0,RD
SUB       A1.7,A1.7,D0.7
SUB       A1.7,A1LbP,A0.7
```

Usage notes

See Operand 2 replace in section 3.1.3

2.3.45. SUBcc Rx.r,Ax.r,Ae.r

Performs a conditional address unit arithmetic subtraction into any destination.

See encoding diagram 3.3.45

Syntax

SUBcc **Dst,Src1,Src2**

Dst	Ax.r | Dx.r | PC.r | RAxx | CT.r | TR.r | TT.r
Src1	Ax.r | CPCx
Src2	Ae.r | CPCe

Operation

```
0:           SUBVal = AUSub(Src1, Src2);
1: <cc> ->   'Dst   = SUBVal;
```

Examples

```
 SUBMI      PCX,CPC1,CPC1
 SUBCS      TXRPT,A0FrP,A0.7
 SUBEQ      RAMX,A1LbP,CPC1
```

Usage notes

See section 2.1.9 Conditional operations.

2.3.46. SUBcc Rx.r,Ax.r,Rx.r

Performs a conditional cross-unit arithmetic subtraction with an address unit as the primary unit.

See encoding diagram 3.3.46

Syntax

SUBcc **Dst,Src1,Src2**

Dst	Ax.r | Dx.r | PC.r | RAxx | CT.r | TR.r | TT.r
Src1	Ax.r | CPCx
Src2	Ax.r | Dx.r | RD

Operation

```
-1:          O2RSrc2 = O2R(Src2);
0:           SUBVal  = AUSub(Src1, O2RSrc2);
1: <cc> ->   'Dst    = SUBVal;
```

Examples

```
 SUBCS     TXMASKI,CPC1,A0FrP
 SUBVC     RAMX,A0FrP,A1LbP
 SUBHI     D1.7,A0.7,D1.7
```

Usage notes

See section 2.1.9 Conditional operations.

See Operand 2 replace in section 3.1.3

2.3.47. SUB Rx.r,Ax.r,#X8

Performs a cross-unit arithmetic subtraction with 8-bit immediate data and an address unit as the primary unit.

See encoding diagram 3.3.47

Syntax

SUB **Dst,Src1,Src2**

Dst	Ax.r | Dx.r | PC.r | RAxx | CT.r | TR.r | TT.r
Src1	Ax.r
Src2	#X8

Operation

```
0:          SUBVal = AUSub(Src1, Src2);
1:          'Dst   = SUBVal;
```

Examples

```
SUB      A0.7,A1.7,#0x7f
SUB      D0.7,A0.7,#0x7f
SUB      TXMASKI,A1.7,#0x7f
```

Usage notes

2.3.48. SUBcc Ae.r,Ax.r,#X8

Performs a conditional address unit arithmetic subtraction with 8-bit immediate data.

See encoding diagram 3.3.48

Syntax

SUBcc **Dst,Src1,Src2**

Dst	Ae.r
Src1	Ax.r | CPCx
Src2	#X8

Operation

```
0:           SUBVal = AUSub(Src1, Src2);
1: <cc> ->   'Dst   = SUBVal;
```

Examples

```
 SUBCS      A1.7,A1.7,#0x80
 SUBLE      A0.7,CPC0,#0x80
```

Usage notes

See section 2.1.9 Conditional operations.

2.3.49. SUB De.r,Dx.r,Rx.r

Performs a cross-unit arithmetic subtraction into a data unit destination.

See encoding diagram 3.3.49

Syntax

SUB[S] **Dst,Src1,Src2**

Dst	De.r
Src1	Dx.r
Src2	Dx.r | Ax.r | RD

Operation

```
-1:         O2RSrc2 = O2R(Src2);
0:          'Dst    = DUSub(Src1, O2RSrc2),
    [S] -> 'Flags   = DUFlags;
```

Examples

```
 SUB        D0.7,D0.7,A0.7
 SUB        D0.7,D0.7,RD
 SUBS       D0.7,D0.7,A1.7
```

Usage notes

See Operand 2 replace in section 3.1.3

2.3.50. SUBcc Rx.r,Dx.r,De.r

Performs a conditional data unit arithmetic subtraction into any destination.
See encoding diagram 3.3.50

Syntax

SUB[S]cc **Dst,Src1,Src2**

Dst	Ax.r | Dx.r | PC.r | RAxx | CT.r | TR.r | TT.r
Src1	Dx.r
Src2	De.r

Operation

```
0:                    SUBVal = DUSub(Src1, Src2),
   ([S] && <cc>) -> 'Flags = DUFlags;
1:          <cc> -> 'Dst   = SUBVal;
```

Examples

```
 SUBCS     A1.7,D0.7,D0Re0
 SUBSMI    RAWX,D0Re0,D0.7
 SUBNE     TTEXEC,D1Re0,D1.7
```

Usage notes

See section 2.1.9 Conditional operations.

2.3.51. SUBcc Rx.r,Dx.r,Rx.r

Performs a conditional cross-unit arithmetic subtraction with a data unit as the primary unit.
See encoding diagram 3.3.51

Syntax

SUB[S]cc **Dst,Src1,Src2**

Dst	Ax.r | Dx.r | PC.r | RAxx | CT.r | TR.r | TT.r
Src1	Dx.r
Src2	Ax.r | Dx.r | RD

Operation

```
-1:                 O2RSrc2 = O2R(Src2);
0:                  SUBVal  = DUSub(Src1, O2RSrc2),
   ([S] && <cc>) -> 'Flags  = DUFlags;
1:          <cc> -> 'Dst    = SUBVal;
```

Examples

```
 SUBGE     A0.7,D0.7,RD
 SUBSGT    RAWX,D0Re0,A0.7
 SUBNV     TXRPT,D1Re0,D0.7
```

Usage notes

See section 2.1.9 Conditional operations.
See Operand 2 replace in section 3.1.3

2.3.52. SUB Rx.r,Dx.r,#X8

Performs a cross-unit arithmetic subtraction with 8-bit immediate data and a data unit as the primary unit.

See encoding diagram 3.3.52

Syntax

SUB[S] **Dst,Src1,Src2**

Dst	Ax.r | Dx.r | PC.r | RAxx | CT.r | TR.r | TT.r
Src1	Dx.r
Src2	#X8

Operation

```
0:          SUBVal = DUSub(Src1, Src2);
    [S] -> 'Flags = DUFlags;
1:          'Dst   = SUBVal;
```

Examples

```
 SUB       A0.7,D0.7,#0x7f
 SUB       D0.7,D1Re0,#0xff
 SUBS      TXMASKI,D1.7,#0x7f
```

Usage notes

2.3.53. SUBcc De.r,Dx.r,#X8

Performs a conditional data unit arithmetic subtraction with 8-bit immediate data.

See encoding diagram 3.3.53

Syntax

SUB[S]cc **Dst,Src1,Src2**

Dst	De.r
Src1	Dx.r
Src2	#X8

Operation

```
0:          <cc>  -> 'Dst   = DUSub(Src1, Src2),
   ([S] && <cc>) -> 'Flags = DUFlags;
```

Examples

```
 SUBCS     D1.7,D1.7,#0x80
 SUBSNV    D0.7,D0.7,#0xff
```

Usage notes

See section 2.1.9 Conditional operations.

2.3.54. TSTcc Dx.r,De.r

Performs a conditional test and updates all the condition flags to indicate the result of an equivalent bit-wise AND operation based on the two input parameters.

See encoding diagram 3.3.54

Syntax

TSTcc **Src1,Src2**

Src1	Dx.r
Src2	De.r

Operation

```
0:          DUAnd(Src1, Src2),
   <cc> ->  'Flags = DUFlags;
```

Examples

```
  TSTCS     D0Re0,D0Re0
  TSTMI     D1Re0,D1.7
```

Usage notes

See section 2.1.8 Condition-flag setting operations.

2.3.55. TSTcc Dx.r,Rx.r

Performs a conditional test and updates all the condition flags to indicate the result of an equivalent bit-wise AND operation based on the two input parameters.

See encoding diagram 3.3.55

Syntax

TSTcc **Src1,Src2**

Src1	Dx.r
Src2	Dx.r | Ax.r | RD

Operation

```
-1:          O2RSrc2 = O2R(Src2);
0:           DUAnd(Src1, O2RSrc2),
   <cc> ->   'Flags = DUFlags;
```

Examples

```
  TSTGT      D0.7,A1.7
  TSTNV      D0Re0,D1.7
  TSTVS      D0Re0,RD
```

Usage notes

See section 2.1.8 Condition-flag setting operations.

See Operand 2 replace in section 3.1.3

2.3.56. TSTcc Dx.r,#X8

Performs a conditional test and updates all the condition flags to indicate the result of an equivalent bit-wise AND operation based on the two input parameters.

See encoding diagram 3.3.56

Syntax

TSTcc **Src1,Src2**

Src1	Dx.r
Src2	#X8

Operation

```
0:          DUAnd(Src1, Src2),
   <cc> ->  'Flags = DUFlags;
```

Examples

```
  TSTLE     D0.7,#0x7f
  TSTGE     D1Re0,#0x7f
```

Usage notes

See section 2.1.8 Condition-flag setting operations.

2.3.57. XOR De.r,Dx.r,Rx.r

Performs a cross-unit bitwise XOR into a data unit destination.

See encoding diagram 3.3.57

Syntax

XOR[S] **Dst,Src1,Src2**

Dst	De.r
Src1	Dx.r
Src2	Dx.r | Ax.r | RD

Operation

```
-1:        O2RSrc2 = O2R(Src2);
0 :        'Dst    = DUXor(Src1, O2RSrc2),
    [S] -> 'Flags  = DUFlags;
```

Examples

```
  XOR       D0.7,D0.7,A0.7
  XOR       D0Re0,D0Re0,D1.7
  XORS      D1.7,D1Re0,RD
```

Usage notes

See Operand 2 replace in section 3.1.3

2.3.58. XORcc Rx.r,Dx.r,De.r

Performs a conditional data unit bitwise XOR into any destination.
See encoding diagram 3.3.58

Syntax

XOR[S]cc **Dst,Src1,Src2**

Dst	Ax.r | Dx.r | PC.r | RAxx | CT.r | TR.r | TT.r
Src1	Dx.r
Src2	De.r

Operation

```
0:         <cc>  -> 'Dst   = DUXor(Src1, Src2),
   ([S] && <cc>) -> 'Flags = DUFlags;
```

Examples

```
  XORNE     A0.7,D1.7,D1.7
  XORCS     RAMX,D1Re0,D1Re0
  XORSEQ    PC,D1.7,D1Re0
```

Usage notes

See section 2.1.9 Conditional operations.

2.3.59. XORcc Rx.r,Dx.r,Rx.r

Performs a conditional cross-unit bitwise XOR with a data unit as the primary unit.

See encoding diagram 3.3.59

Syntax

XOR[S]cc Dst,Src1,Src2

Dst	Ax.r | Dx.r | PC.r | RAxx | CT.r | TR.r | TT.r
Src1	Dx.r
Src2	Dx.r | Ax.r | RD

Operation

```
-1:                 O2RSrc2 = O2R(Src2);
0 :                 XORVal  = DUXor(Src1, O2RSrc2),
    ([S] && <cc>) -> 'Flags  = DUFlags;
1:            <cc> -> 'Dst    = XORVal;
```

Examples

```
  XORGE     GTEXEC,D0Re0,A0.7
  XORGT     RA,D1.7,RD
  XORSGT    TXMASKI,D0Re0,A0.7
  XORSVS    D1.7,D1Re0,A1.7
```

Usage notes

See section 2.1.9 Conditional operations.

See Operand 2 replace in section 3.1.3

2.3.60. XOR Rx.r,Dx.r,#X8

Performs a cross-unit bitwise XOR with 8-bit immediate data with a data unit as the primary unit.
See encoding diagram 3.3.60

Syntax

XOR[S] **Dst,Src1,Src2**

Dst	Ax.r | Dx.r | PC.r | RAxx | CT.r | TR.r | TT.r
Src1	Dx.r
Src2	#X8

Operation

```
0:          XORVal = DUXor(Src1, Src2),
    [S] -> 'Flags = DUFlags;
1:          'Dst   = XORVal;
```

Examples

```
 XOR        A0.7,D0.7,#0x7f
 XOR        D0.7,D1Re0,#0xff
 XORS       TXMASKI,D1.7,#0x7f
```

Usage notes

2.3.61. XORcc De.r,Dx.r,#X8

Performs a conditional data unit bitwise XOR with 8-bit immediate data.

See encoding diagram 3.3.61

Syntax

XOR[S]cc **Dst,Src1,Src2**

Dst	De.r
Src1	Dx.r
Src2	#X8

Operation

```
0:         <cc>  -> 'Dst   = DUXor(Src1, Src2),
   ([S] && <cc>) -> 'Flags = DUFlags;
```

Examples

```
XORGT     D1.7,D1.7,#0xff
XORSNE    D0.7,D0.7,#0x80
```

Usage notes

See section 2.1.9 Conditional operations.

2.4. Data unit extended instructions

2.4.1. ABS De.r,Dx.r

Stores the absolute (i.e. positive only) value of Src into the destination register.

See encoding diagram 3.4.1

Syntax

ABS	**Dst,Src**
Dst	De.r
Src	Dx.r

Operation

```
0:          'Dst   = DUAbs(Src);
```

Examples

```
  ABS       D0.7,D0.7                                                 ;
  ABS       D0.7,D0Re0
  ABS       D1.7,D1.7
```

Usage notes

The condition flags are always changed by this instruction (as if a CMP Src, #0x80000000 had been performed). If the source register contains 0x80000000 the destination will be set to 0x80000000 and the Z flag will be set to indicate that this boundary condition has occurred.

See section 2.1.8 Condition-flag setting operations.

2.4.2. FFB De.r,Dx.r

Indicates the bit position of the first non-sign bit in a register value.

See encoding diagram 3.4.2

Syntax

FFB	**Dst,Src**
Dst	De.r
Src	Dx.r

Operation

```
0:          'Dst = DUFfb(Src);
```

Examples

```
  FFB       D0.7,D0.7                                              ;
  FFB       D0.7,D0Re0
  FFB       D1.7,D1.7
```

Usage notes

If the source register contains no bit changes (i.e. is either 0x00000000 or 0xFFFFFFFF) the result will be 0xFFFFFFFF (-1).

This operation affects the condition flags as if a compare of Src with 0 has taken place, which allows the case where no bit is set to be detected with Dst containing 0 and the Z condition flag being set.

See section 2.1.8 Condition-flag setting operations.

2.4.3. MAX De.r,Dx.r,De.r

Stores the larger of two operands into the destination register.

See encoding diagram 3.4.3

Syntax

MAX	**Dst,Src1,Src2**
Dst	De.r
Src1	Dx.r
Src2	De.r

Operation

```
0:          'Dst = DUMax(Src1, Src2);
```

Examples

```
  MAX       D0.7,D0.7,D0.7
  MAX       D1Re0,D1.7,D1.7
```

Usage notes

The comparison used is a signed greater than or equals compare (i.e. largest signed quantity will be used). The flags are set correctly as per a normal compare (i.e. CMP Src1, Src2).

See section 2.1.8 Condition-flag setting operations.

2.4.4. MIN De.r,Dx.r,De.r

Stores the smaller of two operands into the destination register.

See encoding diagram 3.4.4

Syntax

MIN	**Dst,Src1,Src2**
Dst	De.r
Src1	Dx.r
Src2	De.r

Operation

```
0:          'Dst = DUMim(Src1, Src2);
```

Examples

```
  MIN       D0.7,D0.7,D0.7
  MIN       D1Re0,D1.7,D1.7
```

Usage notes

The comparison used is a signed less than compare (i.e. smallest signed quantity will be used). The flags are set correctly as per a normal compare (i.e. CMP Src1, Src2).

See section 2.1.8 Condition-flag setting operations.

2.4.5. MORT De.r,Dx.r,De.r

Extracts the lower 16 bits of the first source operand, mux them into the even bits of the result, and extracts the lower 16 bits of the second source operand, mux them into the odd bits of the result.

See encoding diagram 3.4.5

Syntax

MORT **Dst,Src1,Src2**

Dst	De.r
Src1	Dx.r
Src2	De.r

Operation

```
0:          MORTSrc1 = Src1 & 0xFFFF,
            MORTSrc2 = Src2 & 0xFFFF,
            'Dst = MORTSrc2(15):MORTSrc1(15):…: MORTSrc2(0):MORTSrc1(0);
```

Examples

```
  MORT      D0.7,D0.7,D0.7
  MORT      D0Re0,D0.7,D0Re0
  MORT      D1Re0,D1.7,D1.7
```

Usage notes

This instruction produces a 32-bit result from two source operands.

This operation affects the condition flags as if a compare of Src1 with Src2 has taken place.

See section 2.1.8 Condition-flag setting operations.

Note: This instruction is not available on HTP 221 cores.

2.4.6. MORT De.r,Dx.r,Rx.r

Extracts the lower 16 bits of the first source operand, mux them into the even bits of the result, and extracts the lower 16 bits of the second source operand, mux them into the odd bits of the result.

See encoding diagram 3.4.6

Syntax

MORT **Dst,Src1,Src2**

Dst	De.r
Src1	Dx.r
Src2	BBx.r | RD

Operation

```
-1:        O2RSrc2 = O2R(Src2);
0:         MORTSrc1 = Src1 & 0xFFFF,
           MORTSrc2 = O2RSrc2 & 0xFFFF,
           'Dst = MORTSrc2(16):MORTSrc1(16):…: MORTSrc2(0):MORTSrc1(0);
```

Examples

```
  MORT      D0.7,D0.7,A0.7
  MORT      D0Re0,D0.7,RD
  MORT      D1Re0,D1.7,A1.7
```

Usage notes

This instruction produces a 32-bit result from two source operands.

This operation affects the condition flags as if a compare of Src1 with Src2 has taken place.

See section 2.1.8 Condition-flag setting operations.

See Operand 2 replace in section 3.1.3

Note: This instruction is not available on HTP 221 cores.

2.4.7. NMIN De.r,Dx.r,De.r

Evaluates the number of left shifts required to represent the value in Src1 fully as a signed number (NORM) and then performs a MIN test between this result and the value in Src2.

See encoding diagram 3.4.7

Syntax

NMIN **Dst,Src1,Src2**

Dst	De.r
Src1	Dx.r
Src2	De.r

Operation

```
0:          'Dst = DUNmin(Src1, Src2);
```

Examples

```
  NMIN      D0.7,D0.7,D0.7
  NMIN      D0Re0,D0.7,D0Re0
  NMIN      D1Re0,D1.7,D1.7
```

Usage notes

A combination of the NORM and MIN instructions. This evaluates the number of left shifts required to represent the value in Src1 fully as a signed number and then performs a MIN test between this result and the value in Src2. The flags are affected as per the NORM instruction.

See section 2.1.8 Condition-flag setting operations.

2.4.8. NORM De.r,Dx.r

Returns the number of left shifts required to represent the value fully as a signed number.
See encoding diagram 3.4.8

Syntax

NORM	**Dst,Src**
Dst	De.r
Src1	Dx.r

Operation

```
0:          'Dst = DUNorm(Src);
```

Examples

```
  NORM       D0.7,D0.7
  NORM       D1Re0,D1.7
```

Usage notes

The condition flags are always changed by this instruction, as if a CMP Src, #0 had been performed. See section 2.1.8 Condition-flag setting operations.

2.4.9. VPACK De.r,Dx.r,De.r

Extracts the lower 16 bits of the first and second source operand, packs them into a 32-bit result with the lower 16 bits from the first source operand and the upper 16 bits from the second source operand.

See encoding diagram 3.4.9

Syntax

VPACK **Dst,Src1,Src2**

Dst	De.r
Src1	Dx.r
Src2	De.r

Operation

```
0:          VPACKSrc1 = Src1 & 0xFFFF;
            VPACKSrc2 = Src2 & 0xFFFF;
            'Dst = (VPACKSrc2 << 16) | VPACKSrc1;
```

Examples

```
  VPACK     D1.7,D1Re0,D1.5
  VPACK     D0.2,D0.1,D0Re0
```

Usage notes

This operation affects the condition flags as if a compare of Src1 with Src2 has taken place.

See section 2.1.8 Condition-flag setting operations.

Note: This instruction is not available on HTP 221 cores.

2.4.10. VPACK De.r,Dx.r,Rx.r

Extracts the lower 16 bits of the first and second source operand, packs them into a 32-bit result with the lower 16 bits from the first source operand and the upper 16 bits from the second source operand.

See encoding diagram 3.4.10

Syntax

VPACK **Dst,Src1,Src2**

Dst	De.r
Src1	Dx.r
Src2	Do.r | Ax.r | RD

Operation

```
-1:         O2RSrc2    = O2R(Src2);
0:          VPACKSrc1 = Src1 & 0xFFFF,
            VPACKSrc2 = O2RSrc2 & 0xFFFF,
            'Dst       = (VPACKSrc2 << 16) | VPACKSrc1;
```

Examples

```
  VPACK     D1.7,D1Re0,A1.5
  VPACK     D1.7,D1Re0,D0.5
  VPACK     D0.2,D0.1,RD
```

Usage notes

This operation affects the condition flags as if a compare of Src1 with Src2 has taken place.

See section 2.1.8 Condition-flag setting operations.

See Operand 2 replace in section 3.1.3

Note: This instruction is not available on HTP 221 cores.

2.5. Shift instructions

2.5.1. SHIFT De.r,Dx.r,De.r|#X5

Data unit logical or arithmetic shift left or right allowing the use of signed shift specifiers.

See encoding diagram 3.5.1

Syntax

LSL|LSR|ASR[S] **Dst,Src1,Src2**

Dst	De.r
Src1	Dx.r
Src2	De.r | #X5

Operation

```
0:          'Dst   = DULsl|DULsr|DUAsr(Src1, Src2),
   [S] ->   'Flags = DUFlags;
```

Examples

```
  ASR       D0.7,D0Re0,D0.7
  ASRS      D1.7,D1.7,D1.7
  LSLS      D1Re0,D1.7,D1Re0
  LSR       D0Re0,D0Re0,D0.7

  ASR       D1.7,D1.7,#0xf
  ASRS      D1Re0,D1.7,#0x10
  LSLS      D1.7,D1Re0,#0x1f
  LSR       D1.7,D1Re0,#0x1f
```

Usage notes

Only the bottom five bits of Src2 are used (other bits set in a register operand do nothing) and used as an unsigned number.

ASR duplicates the top-most bit of signed values during a right shift, whereas LSR shifts in zeroes for unsigned values; left shifts via LSL can be applied to any data type shifting in zeros.

2.5.2. SHIFT Rx.r,Dx.r,De.r|#X5

Cross-unit logical or arithmetic shift left or right allowing the use of signed shift specifiers.

See encoding diagram 3.5.2

Syntax

LSL|LSR|ASR[S] **Dst,Src1,Src2**

Dst	Ax.r | Do.r | PC.r | RAxx | CT.r | TR.r | TT.r
Src1	Dx.r
Src2	De.r | #X5

Operation

```
0:          'SHIFTVal = DULsl|DULsr|DUAsr(Src1, Src2),
   [S] ->   'Flags    = DUFlags;
1:          'Dst      = SHIFTVal;
```

Examples

```
  ASR       A0.7,D1Re0,D1.7
  ASRS      PC,D0.7,D0.7
  LSRS      RABZ,D0.7,D0Re0
  LSL       GTEXEC,D0.7,D0.7

  ASR       D0.7,D1.7,#0xf
  LSL       PC,D1.7,#0x10
  LSR       RA,D0Re0,#0x1f
  LSRS      TXMASKI,D1.7,#0x1f
```

Usage notes

Only the bottom five bits of Src2 are used (other bits set in a register operand do nothing) and used as an unsigned number.

ASR duplicates the top-most bit of signed values during a right shift, whereas LSR shifts in zeroes for unsigned values; left shifts via LSL can be applied to any data type shifting in zeros.

2.5.3. SHIFTcc De.r,Dx.r,De.r|#X5

Conditional logical or arithmetic shift left or right allowing the use of signed shift specifiers.

See encoding diagram 3.5.3

Syntax

LSL|LSR|ASR[S]cc **Dst,Src1,Src2**

Dst	De.r
Src1	Dx.r
Src2	De.r | #X5

Operation

```
0: <cc>          -> 'Dst   = DULsl|DULsr|DUAsr(Src1, Src2),
   ([S] && <cc>) -> 'Flags = DUFlags;
```

Examples

```
  ASRMI     D0Re0,D0Re0,D0.7
  LSLSCS    D0Re0,D0.7,D0.7
  LSRLT     D0.7,D0.7,D0Re0

  ASRLE     D0Re0,D0Re0,#0x1f
  ASRSLT    D0Re0,D0.7,#0x10
  LSRSLE    D1.7,D1Re0,#0x10
```

Usage notes

Only the bottom five bits of Src2 are used (other bits set in a register operand do nothing) and used as an unsigned number.

ASR duplicates the top-most bit of signed values during a right shift, whereas LSR shifts in zeroes for unsigned values; left shifts via LSL can be applied to any data type shifting in zeros.

See section 2.1.9 Conditional operations.

2.6. Long general purpose instructions

2.6.1. BEXx De.r,Dx.r

Performs a 32-bit or 64-bit byte endian exchange.

See encoding diagram 3.6.1

Syntax

BEX<S>D **Dst,Src**

Dst	De.r
Src	Dx.r

BEX<S>L **Dst,Src**

Dst	Do.r
Src	Dx.r

Operation

BEXD

```
0:          'Dst = DUBex(Src);
```

BEXL

```
0:          BexVal  = DUBex(Src),
            OBexVal = DUBex(Src.o);
1:          'Dst    = OBexVal,
            'Dst.o  = BexVal;
```

Examples

```
  BEXD      D0Re0,D0.7
  BEXSD     D0Re0,D0Re0

  BEXL      D0Re0,D1.7
  BEXSL     D1.7,D0.7
```

Usage notes

One of **`[D]`** or **`[L]`** must be specified.

`[D]` - for the 32-bit operation the four bytes in a 32-bit data unit register are reordered from MSB-3-2-1-0-LSB to MSB-0-1-2-3-LSB.

`[L]` - for the 64-bit operation the instruction performs the same function in each of the two source registers and then swaps the registers to achieve a 64-bit endian swap.

Flags may be set to reflect the value of the primary 32-bit part of the result generated.

See section 2.1.8 Condition-flag setting operations.

2.6.2. CALL BBx.r,#X16

Performs a call to a statically defined sub-routine using an absolute address offset.

See encoding diagram 3.6.2

Syntax

CALL **Link,Value**

Link	BBx.r
Value	#X16

Operation

```
0: 'Link = PCUNext(PC),
   'PC   = PCUAdd(Link, Value);
```

Examples

```
  CALL      D1RtP,#LO(Label)
  CALL      D0Re0,#-0x4
```

Usage notes

This specialised exchange instruction optimises the code needed to execute statically defined sub-routines. This instruction implicitly targets the PC register.

Conventionally D1RtP (which is defined in machine.inc) should be used for **Link** when calling normal sub-routines.

A call to a sub-routine, can be implemented with the instruction sequence:

```
MOVT   D1RtP, #HI(__SymTable._printf)              ; __SymTable. -> global label
CALL   D1RtP, #LO(__SymTable._printf)
```

2.6.3. CPRLcc BBx.r,BBo.r,CP.n

Performs a conditional 64-bit read from a coprocessor read port into a pair of registers.
See encoding diagram 3.6.3

Syntax

CPRLcc **Dst1,Dst2,Src**

Dst1	BBx.r
Dst2	BBo.r
Src	CP.n

Operation

```
0:         Size = L,
1:  <cc> -> 'Dst1, 'Dst2 = CPRead(Src, Size);
```

Examples

```
  CPRLLT    A1.7,A0.7,CP.7
  CPRLLT    D0.7,D1.7,CP.0
  CPRLNV    A0FrP,A1.7,CP.7
```

Usage notes

Dst1 receives bits 31 to 0 and Dst2 receives bits 63 to 32.
Data from the co-processor is discarded if the condition is false.
See section 2.1.9 Conditional operations.

2.6.4. CPRR [BBxUaRoPp],CP.n

Performs a repeating read from a coprocessor read port into memory.

See encoding diagram 3.6.4

Syntax

CPR[D|L]R **[Dst],Src**

[Dst]	[BBxUaRoPp]
Src	CP.n

Operation

```
n+0:   Addr = BUAddr(Dst),
       Size = [D|L],
       Data = CPRead(Src, Size),
       'TXRPT = TXRPT - 1;
n+2:   MemStore(Addr, Size, Data);
```

Examples

```
  CPRDR     [A1LbP++A1LbP],CP.0
  CPRLR     [D0.7+D0Re0],CP.0
  CPRDR     [D1Re0++D1Re0],CP.7
  CPRLR     [A0.7++A0.7],CP.7
```

Usage notes

CPRR repeats the number of times specified in control register TXRPT.

See section 2.1.4 Address modes.

2.6.5. CPRxcc Ux.r,CP.n

Performs a conditional read from a coprocessor port any destination unit register.

See encoding diagram 3.6.5

Syntax

CPR[B|W|D]cc **Dst,Src**

Dst	BBx.r | PC.r | CT.r | TR.r | TT.r
Src	CP.n

Operation

```
0:           Size = [B|W|D],
             Data = CPRead(Src, Size);
1: <cc> ->   'Dst = Data;
```

Examples

```
  CPRBLT     A1.7,CP.7
  CPRDHI     PC,CP.0
  CPRDHI     TXMASKI,CP.7
  CPRDCS     TXSTAT,CP.0
  CPRDNV     GTEXEC,CP.0
  CPRWGT     D0Re0,CP.0
```

Usage notes

One of **`[B]`**, **`[W]`** or **`[D]`** must be specified.

PC.r | CT.r | TR.r | TT.r can only be read using a 32-bit **`[D]`** instruction.

See section 2.1.9 Conditional operations.

2.6.6. CPRxcc [BBx.r],CP.n

Performs a conditional read from a coprocessor read port into memory.

See encoding diagram 3.6.6

Syntax

CPR[B|W|D|L]cc **[Dst],Src**

[Dst]	[BBx.r]
Src	CP.n

Operation

```
0:          Addr = BUAddr(Dst),
            Size = [B|W|D|L],
            Data = CPRead(Src, Size);
2: <cc> -> MemStore(Addr, Size, Data);
```

Examples

```
  CPRBEQ    [D1Re0],CP.0
  CPRDEQ    [A0FrP],CP.0
  CPRWVS    [D1.7],CP.0
  CPRLGE    [A1LbP],CP.7
```

Usage notes

One of **`[B]`**, **`[W]`**, **`[D]`** or **`[L]`** must be specified.

See section 2.1.9 Conditional operations.

2.6.7. CPWL CP.n,BBx.r,BBo.r|#X16

Performs a 64-bit write to a coprocessor write port from a pair of registers or a 16-bit value.
See encoding diagram 3.6.7

Syntax

CPWL **Dst,Src1,Src2**

Dst	CP.n
Src1	BBx.r
Src2	BBo.r|#X16

Operation

```
0:   Size = L,
     Data, Data.o = Src1, Src2;
2:   CPWrite('Dst, Size, Data);
```

Examples

```
  CPWL       CP.0,A1.7,A0.7
  CPWL       CP.7,D0.7,D1.7
  CPWL       CP.0,A1LbP,#0xffff
  CPWL       CP.0,D0.7,#0x7fff
```

Usage notes

`[L]` can be omitted as it is implied by having two source operands.

2.6.8. CPWL CP.n,[BBxUaRoPp|BBxUaS8Pp]

Performs a repeating 64-bit write to a coprocessor write port from memory using the read pipeline.
See encoding diagram 3.6.8

Syntax

CPWL[R] **Dst,[Src]**

Dst	CP.n
[Src]	[BBxUaRoPp|BBxUaS8Pp]

Operation

```
n+0:                           Size    = L,
                               DstData = RD,
                               SrcAddr   = BUAddr(Src),
     ([R] && (TXRPT != 0)) -> 'TXRPT = TXRPT - 1;
n+2:                           CPWrite('Dst, Size, DstData);
                               SrcData   = RAMov(SrcAddr, Size);
n+3:                           'RD = RAMod(SrcData, RA, SrcSize);
```

Examples

```
  CPWL      CP.0,[A0.7+A0FrP]
  CPWL      CP.0,[D1.7+D1.7++]
  CPWLR     CP.7,[A0FrP++A0FrP]
  CPWLR     CP.7,[D1Re0+D1.7++]

  CPWL      CP.0,[A0.7+#-0x1++]
  CPWL      CP.7,[A1LbP+#0x7f]
  CPWLR     CP.0,[A0FrP+#-0x80]
  CPWLR     CP.7,[D1Re0+#-0x1]
```

Usage notes

`[R]` specifies a repeating instruction that repeats the number of times specified in control register TXRPT. Without this option the instruction is issued once unconditionally.

`[L]` can be omitted as it is implied by having two source operands.

See section 2.1.4 Address modes.

2.6.9. CPXL CP.n,CP.n

Performs a repeating 64-bit transfer between coprocessors.
See encoding diagram 3.6.9

Syntax

CPXL[R] **Dst,Src**

Dst	CP.n
Src	CP.n

Operation

CPXL

```
n+0:                              Size = L,
                                  Data = CPRead(Src, Size),
     ([R] && (TXRPT != 0))_-> 'TXRPT = TXRPT - 1;
n+2:                              CPWrite('Dst, Size, Data);
```

Examples

```
  CPXL      CP.0,CP.0
  CPXL      CP.7,CP.0
  CPXLR     CP.0,CP.7
  CPXLR     CP.7,CP.7
```

Usage notes

[R] specifies a repeating instruction that repeats the number of times specified in control register TXRPT. If [R] is omitted the operation is issued only once.

2.6.10. DEFRcc BBx.r,TR.r

Reads and clears deferred error states via the trigger system.

See encoding diagram 3.6.10

Syntax

DEFRcc **Dst,Src**

Dst	BBx.r
Src	TR.r

Operation

```
0:   XUData = ((TXDEFR & 0xFFFF0000) | (TXSTAT & 0x0000FFFF),
     'TXDEFR = TXDEFR & 0x0000FFFF;
1:   'Dst = XUData;
```

Examples

```
  DEFRLT     A0.0,TXSTAT
  DEFRGE     A1.7,TXPOLLI
  DEFRNV     D0.7,TXSTATI
  DEFRLE     D1.3,TXPOLL
```

Usage notes

TR.r is one of TXSTAT, TXSTATI, TXPOLL or TXPOLLI.
This instruction reads deferred error state (as per the TXDEFR register) appropriate to the processing level specified, so a DEFR of TXSTAT or TXPOLL returns the background deferred error state and a DEFR of TXSTATI or TXPOLLI returns the interrupt deferred error state.

DEFR also clears all the deferred error state it read when the source register is either TXSTAT or TXSTATI.

If TXPOLL or TXPOLLI is the source register then the deferred error data for that processing level is read without waiting for any possible deferred errors in the pipeline to reach the TXDEFR register. If TXSTAT or TXSTATI is the source register the instruction waits for any possible deferred errors in the pipeline to reach the TXDEFR register and waits for any possible risk of pipeline rewind to pass before the operation is issued.

See section on Deferred Triggers in Architecture Overview TRM.

See section 2.1.9 Conditional operations.

2.6.11. GETL BBx.r,BBo.e,[BBa.r+#S12]

Loads a pair of registers from memory using register + 12-bit signed word offset address. The base address register number must be 0 or 1 to gain access to the extended offset range.

See encoding diagram 3.6.11

Syntax

GETL **Dst1,Dst2,[Src]**

Dst1	BBx.r
Dst2	BBo.e
[Src]	[BBa.0|1+#S12]

Operation

```
0:      Addr         = BUAddr(Src),
        Size         = L;
2:      Data.o, Data = MemLoad(Addr, Size);
3:      'Dst1, 'Dst2 = Data, Data.o;
```

Examples

```
  GETL      A1.7,A0.7,[A1LbP+#-0x8]
  GETL      A0.7,A1.7,[D1Re0+#-0x8]
  GETL      D0.7,D1.7,[A1LbP+#0x3ff8]
  GETL      D0.7,D1.7,[D1Re0+#-0x4000]
```

Usage notes

The same address/data unit may be used to generate the memory address and specify the register to be loaded.

See section 2.1.4 Address modes.

2.6.12. GETx BBx.r,[BBa.r+#S12]

Loads a single register from memory using register + signed 12-bit word offset address. The base address register number must be 0 or 1 to gain access to the extended offset range.

See encoding diagram 3.6.12

Syntax

GET[B|W|D] **Dst,[Src]**

Dst	BBx.r
[Src]	[BBa.0|1+#S12]

Operation

```
0:          Addr   = BUAddr(Src),
            Size   = [B|W|L];
2:          Data   = MemLoad(Addr, Size);
3:          'Dst   = Data;
```

Examples

```
  GETB      A0.7,[A1LbP+#0x7ff]
  GETD      D0.7,[D0Re0+#-0x2000]
  GETW      D0.7,[A1LbP+#-0x1000]
```

Usage notes

The same address/data unit may be used to generate the memory address and specify the register to be loaded.

See section 2.1.4 Address modes.

2.6.13. JUMP BBx.r,#X16

Performs a jump to any global statically defined location. This instruction implicitly targets the PC register.

See encoding diagram 3.6.13

Syntax

JUMP **Src1,Src2**

Src1	BBx.r
Src2	#X16

Operation

```
0:          BBVal = BBAdd(Src1, Src2);
1:          'PC = BBVal;
```

Examples

```
  JUMP      A1LbP,#LO(Label)
  JUMP      A1LbP,#0x7fff
  JUMP      D0.7,#0xffff
  JUMP      D0.7,#0x8000
```

Usage notes

A more specialised **`MOV PC,xxx`**, this instruction optimises the code needed to jump to any global statically defined location. Using the JUMP instruction a static jump can be implemented with the following instruction sequence:

```
MOVT    A1.2, #HI(__SymTable._myfunc)         ; __SymTable. -> global label
JUMP    A1.2, #LO(__SymTable._myfunc)
```

2.6.14. KICKcc Ux.r,TR.r

Conditionally updates the destination register with 16-bits of remaining kick count and the active triggers (where only the kick trigger will be active).

See encoding diagram 3.6.14

Syntax

KICKcc	**Dst,Src**

Dst	BBx.r | CT.r |PC.r |TR.r | RA* | RD	
Src	TR.r	(TXSTAT, TXSTATI or TXPOLL, TXPOLLI)

Operation

```
0:                      Delta       = ( KICK_COUNT != 0 ) ? 1 : 0,
   (KICK_COUNT != 0) -> ‘KICK_COUNT = KICK_COUNT - Delta,
               XUVal       = ((KICK_COUNT – Delta) << 16) | (Src & 0xFFFF);
1:              <cc> -> 'Dst        = XUVal;
```

Examples

```
  KICKGT    A0FrP,TXSTAT
  KICKHI    D1Re0,TXSTATI     ; NB: Valid when executing TXSTATUS.ISTAT=1
  KICK      D0Re0,TXPOLLI     ; NB: Valid when executing TXSTATUS.ISTAT=1
```

Usage notes

This instruction is a specialised form of a blocking trigger read, which only responds to the kick trigger.

To allow the handling of interactions between separate threads to be easily managed, the Meta core incorporates the concept of software 'kick' events. Kick events are accumulated inside the trigger unit and atomically decremented when a blocked trigger read instruction, that has the kick set as active in the mask, is issued.

This instruction is a short-hand representation for a blocking trigger read that only has the kick active in the trigger mask. Using this instruction allows a thread to interact quickly with its kick accumulator without having to manipulate its trigger mask first.

If a condition is specified, then this only effects the updating of the destination register specified. A KICKNV instruction still waits for a KICK to occur even though it never effects the destination register concerned.

If Src is TXPOLL or TXPOLLI this removes the blocking behaviour and the instruction may proceed and return a KICK_COUNT value of zero.

See section 2.1.9 Conditional operations.

2.6.15. LNKGETL BBx.r,BBo.e,[BBa.r]

Performs a 64-bit linked load from memory. Part of a special linked test and set operation, used in conjunction with LNKSET.

See encoding diagram 3.6.15

Syntax

LNKGETL **Dst1,Dsr2,[Src]**

Dst1	BBx.r
Dst2	BBo.e
[Src]	[BBa.r]

Operation

```
0:              Addr  = BUAddr(Src),
                Size  = L;
2:      Data, Data.o  = MemLoad(Addr, Size),
            LinkFlag  = SetLinkFlag(Size);
3:      'Dst1, 'Dst2  = Data, Data.o;
```

Examples

```
  LNKGETL   D1.1,D0.1,[A1.7]
  LNKGETL   A0.3,A1.3,[D1.0]
```

Usage notes

This instruction reads the data that is to be tested and associates some state with the threads flow of instructions so that the linked set is only completed if that sequence has not been interrupted.

To perform an interruptible 32-bit 'test and set' operation these instructions, the code would be:

```
               ;  D1Ar1 is an address to perform Test/Set logic on using the mask in D0Ar2.

TestAndSet:    LNKGETL D0Re0,D1Re0,[D1Ar1]    ; Perform a linked read
               TST     D0Re0,D0Ar2            ; Wait for the result and check the mask
               TSTZ    D0Re0,D1Ar3            ;    extended to 64-bits
               ORZ     D0Re0,D0Re0,D0Ar2      ; Set the mask if currently clear
               ORZ     D1Re0,D0Re0,D1Ar3      ;    extended to 64-bits
               LNKSETLZ [D1Ar1],D0Re0,D1Re0  ; Perform a linked write if previously clear
               DEFR    D1Re0,TXSTAT           ; Wait for and get deferred error response
               LSRZ D1Re0, D1Re0, #24         ; Get bus error summary
               ANDZ D1Re0, D1Re0, #0x3f
               CMPZ D1Re0, #2                 ; 0x2 indicates LNKSET success
               BNZ     TestAndSet             ; Go back if Z flag is lost at any point
;
; Note local cache state must be flushed/updated before accessing related
; data items after the critical access required is won. Global cache state will remain
; coherent if the core is configured with this option.
;
```

2.6.16. LNKGETx BBx.r,[BBa.r]

Performs a linked load from memory. Part of a special linked test and set operation, used in conjunction with LNKSET.

See encoding diagram 3.6.16

Syntax

LNKGET[B|W|D] **Dst,[Src]**

Dst	BBx.r
[Src]	[BBa.r]

Operation

```
0:      Addr     = BUAddr(Src),
        Size     = [B|W|D];
2:      Data     = MemLoad(Addr, Size),
        LinkFlag = SetLinkFlag(Addr, Size);
3:      'Dst     = Data;
```

Examples

```
  LNKGETB   D1.1,[A1.7]
  LNKGETW   A0.3,[D1.0]
  LNKGETD   A0.1,[A1.1]
```

Usage notes

One of **[B]**, **[W]** or **[D]** must be specified.

This instruction reads the data that is to be tested and associates some state with the threads flow of instructions so that the linked set is only completed if that sequence has not been interrupted.

To perform an interruptible 32-bit 'test and set' operation these instructions, the code would be:

```
               ;  D1Ar1 is an address to perform Test/Set logic on using the mask in D0Ar2.

TestAndSet:    LNKGETD D0Re0,[D1Ar1]          ; Perform a linked read
               TST     D0Re0,D0Ar2            ; Wait for the result and check the mask
               ORZ     D0Re0,D0Re0,D0Ar2      ; Set the mask if currently clear
               LNKSETDZ [D1Ar1],D0Re0         ; Perform a linked write if previously clear
               DEFR    D1Re0,TXSTAT           ; Wait for and get deferred error response
               LSRZ D1Re0, D1Re0, #24         ; Get bus error summary
               ANDZ D1Re0, D1Re0, #0x3f
               CMPZ D1Re0, #2                 ; 0x2 indicates LNKSET success
               BNZ     TestAndSet             ; Go back if Z flag is lost at any point
;
; Note local/global cache state must be flushed/updated before accessing related
; data items after the critical access required is won.
;
```

2.6.17. LNKSETLcc [BBa.r],BBx.r,BBo.e

Performs a 64-bit linked store to memory. Part of a special linked test and set operation, used in conjunction with LNKGET.

See encoding diagram 3.6.17

Syntax

LNKSETLcc **[Dst],Src1,Src2**

[Dst]	[BBa.r]
Src1	BBx.r
Src2	BBo.e

Operation

```
0:                         Addr   = BUAddr(Dst),
                           Size   = L,
                           Data   = Src1, Src2;
2: (LinkFlag && <cc>) -> MemStore(Addr, Size, Data),
                           'LinkFlag = 0;
```

Examples

```
  LNKSETLZ  [D1.1],A1.7,A0.7
  LNKSETLZ  [A0.3],D1.0, D0.0
  LNKSETLZ  [A0.1],A1.1,A0.1
```

Usage notes

See section 2.1.9 Conditional operations.

One of **[B]**, **[W]** or **[D]** must be specified.

This instruction writes the data that is to be set provided the condition is met and the program sequence (since the related LNKGET) has not been interrupted.

To perform an interruptible 32-bit ‘test and set’ operation these instructions, the code would be:

```
               ;  D1Ar1 is an address to perform Test/Set logic on using the mask in D0Ar2.

TestAndSet:    LNKGETD D0Re0,[D1Ar1]          ; Perform a linked read
               TST     D0Re0,D0Ar2            ; Wait for the result and check the mask
               ORZ     D0Re0,D0Re0,D0Ar2      ; Set the mask if currently clear
               LNKSETDZ [D1Ar1],D0Re0         ; Perform a linked write if previously clear
               DEFR    D1Re0,TXSTAT           ; Wait for and get deferred error response
               LSRZ D1Re0, D1Re0, #24         ; Get bus error summary
               ANDZ D1Re0, D1Re0, #0x3f
               CMPZ D1Re0, #2                 ; 0x2 indicates LNKSET success
               BNZ     TestAndSet             ; Go back if Z flag is lost at any point
               ;
; Note local/global cache state must be flushed/updated before accessing related
; data items after the critical access required is won.
;
```

2.6.18. LNKSETxcc [BBa.r],Ux.r

Performs a linked store to memory. Part of a special linked test and set operation, used in conjunction with LNKGET.

See encoding diagram 3.6.18

Syntax

LNKSET[B|W|D] **[Dst],Src**

[Dst]	[BBa.r]
Src	BBx.r | PC.r | CT.r | TR.r

Operation

```
0:                        Addr     = BUAddr(Dst),
                          Size     = [B|W|D],
                          Data     = Src;
2: (LinkFlag && <cc>) -> MemStore(Addr, Size, Data),
                          LinkFlag = 0;
```

Examples

```
  LNKSETBZ  [D1.1],A1.7
  LNKSETWZ  [A0.3],PCX
  LNKSETDZ  [A0.1],TXSTATI
```

Usage notes

See section 2.1.9 Conditional operations.

One of **[B]**, **[W]** or **[D]** must be specified.

This instruction writes the data that is to be set provided the condition is met and the program sequence (since the related LNKGET) has not been interrupted.

To perform an interruptible 32-bit ‘test and set’ operation these instructions, the code would be:

```
               ;  D1Ar1 is an address to perform Test/Set logic on using the mask in D0Ar2.

TestAndSet:    LNKGETD D0Re0,[D1Ar1]          ; Perform a linked read
               TST     D0Re0,D0Ar2            ; Wait for the result and check the mask
               ORZ     D0Re0,D0Re0,D0Ar2      ; Set the mask if currently clear
               LNKSETDZ [D1Ar1],D0Re0         ; Perform a linked write if previously clear
               DEFR    D1Re0,TXSTAT           ; Wait for and get deferred error response
               LSRZ D1Re0, D1Re0, #24         ; Get bus error summary
               ANDZ D1Re0, D1Re0, #0x3f
               CMPZ D1Re0, #2                 ; 0x2 indicates LNKSET success
               BNZ     TestAndSet             ; Go back if Z flag is lost at any point
               ;
; Note local/global cache state must be flushed/updated before accessing related
; data items after the critical access required is won.
;
```

2.6.19. MOV CT.r,#I16

Primes a control unit register with 16-bit data.

See encoding diagram 3.6.19

Syntax

MOV	**Dst,Src**
Dst	CT.r
Src	#I16

Operation

```
0:          XUVal = XUMov(Src);
2:          'Dst  = XUVal;
```

Examples

```
  MOV       TXIDLECYC,#0
  MOV       TXRPT,#40
```

Usage notes

Updates to a number of CT registers such as TXMODE interlock further execution until they are fully completed causing a three cycle stall event to avoid complex interactions between CT setup changes and further execution.

2.6.20. MOVx RAxx,[BBa.r+#S12]

Issues a read address into the read pipeline using a base register plus a 12-bit signed word immediate offset. The base address register number must be 0 or 1 to gain access to the extended offset range.

See encoding diagram 3.6.20

Syntax

MOV[B|W|D|L] **Dst,Src**

Dst	RA.x
Src	[BBa.0|1+#S12]

Operation

```
0:          Addr  = BUAddr(Src),
            Size  = [B|W|D|L];
2:          Data  = RAMov(Addr, Size);
3:          'RD   = RAMod(Data, Src, Size);
```

Examples

```
  MOVB      RA,[A0FrP+#-0x1]
  MOVB      RAMX,[D0Re0+#0x7ff]
  MOVW      RAM8X,[A0FrP+#-0x1000]
  MOVB      RAM8X32,[A0FrP+#0x7ff]
  MOVL      RABZ,[A0FrP+#-0x8]
```

Usage notes

2.6.21. MOVLcc BBx.r,BBo.r,RD

Performs a 64-bit data transfer from memory to a pair of address or data unit register destinations.
See encoding diagram 3.6.21

Syntax

MOVLcc **Dst1,Dst2,Src**

Dst1	BBx.r
Dst2	BBo.r
Src	RD

Operation

```
0:          XUVal     = XUMov(Src);
1: <cc> ->  Dst1,Dst2 = XUVal;
```

Examples

```
  MOVLNV    A0.7,A1.7,RD
  MOVLLE    A1.7,A0.7,RD
  MOVLLE    D0.7,D1.7,RD
  MOVLLT    D0.7,D1.7,RD
  MOVLMI    D1.7,D0.7,RD
  MOVLLT    A1LbP,A0.7,RD
```

Usage notes

Dst1 and Dst2 must be registers in opposite units of the same type.
Dst1 gets the low 32-bits of data, the upper 32-bits go to Dst2.
See section 2.1.9 Conditional operations.

2.6.22. MOVxcc Ux.r,RD

Performs an 8, 16 or 32-bit data transfer from memory to any register destination.

See encoding diagram 3.6.22

Syntax

MOV[B|W|D]cc **Dst,Src**

Dst	BBx.r | CT.r |TR.r | TT.r | PC.r | RAxx
Src	RD

Operation

```
0:          XUVal = XUMov(Src);
1:   <cc> -> 'Dst  = XUVal;
```

Examples

```
  MOVBGT     A0.7,RD
  MOVDNV     D0.7,RD
  MOVDCS     TTREC,RD
  MOVDNV     RAMX,RD
  MOVDNV     TXSTAT,RD
```

Usage notes

Size **`[B|W|D]`** must be specified.

See section 2.1.9 Conditional operations.

2.6.23. MOVcc Ux.r,Uy.r

Performs a conditional cross-unit MOV.

See encoding diagram 3.6.23

Syntax

MOVcc **Dst,Src**

Dst	Dx.r | Ax.r | PC.r | CT.r | TR.r | TT.r | RA.x
Src	Dx.r | Ax.r | PC.r | CT.r | TR.r | TT.r

Operation

```
0:          XUVal = XUMov(Src);
1: <cc> -> 'Dst   = XUVal;
```

Examples

```
  MOVCS     A1.7,TXENABLE
  MOVLE     PCX,D1.7
  MOVVC     D1Re0,A1LbP
  MOVEQ     TXENABLE,A0.7
  MOVEQ     TXENABLE,A1LbP
  MOVGE     D0.7,PCX
  MOVGE     D0.7,TXMASKI
  MOVVC     RAMX,A0FrP
  MOVVC     RAWX,A0FrP
  MOVVC     RAWX,D1.7
```

Usage notes

See section 2.1.9 Conditional operations.

When Dst is RA.x, Src is restricted to Dx.r | Ax.r. This form allows the value of a register to be issued as an address into the read pipeline:

```
  MOVcc RA*, Src              ; RA* -> RA|RABX|RABZ|RAWX|RAWZ|RADX|RADZ|RAM8X|RAM8X32
```

2.6.24. RTDW De.r,Dx.r

Performs a half-word swap - swap top 16-bits of a 32-bit source register with the bottom 16-bits of that register, and write to the 32-bit destination..

See encoding diagram 3.6.24

Syntax

RTD[S]W **Dst,Src**

Dst	De.r
Src	Dx.r

Operation

```
0:          'Dst   = ((Src & 0xFFFF) << 16) | (Src >> 16),
    [S] -> 'Flags = DUFlags;
```

Examples

```
  RTDSW     D0.7,D0Re0
  RTDSW     D1Re0,D1.7
  RTDW      D0Re0,D0.7
  RTDW      D1.7,D1.7
```

Usage notes

The **`W`** modifier is always required.

2.6.25. SETL [BBa.r+#S12],BBx.r,BBo.e

Stores a pair of registers to memory using register plus signed 12-bit word offset address. The base address register number must be 0 or 1 to gain access to the extended offset range.

See encoding diagram 3.6.25

Syntax

SETL **[Dst],Src1,Src2**

[Dst]	[BBa.0|1+#S12]
Src1	BBx.r
Src2	BBo.e

Operation

```
0:              Addr   = BUAddr(Dst),
                Size   = L,
                Data   = Src1, Src2;
2:              MemStore(Addr, Size, Data);
```

Examples

```
  SETL      [A0FrP+#-0x4000],D1Re0,D0Re0
  SETL      [A0FrP+#0x3ff8],A1LbP,A0FrP
  SETL      [D0Re0+#-0x4000],D1Re0,D0Re0
  SETL      [D1Re0+#-0x8],A1LbP,A0FrP
```

Usage notes

The same address/data unit may be used to generate the memory address and specify the register to be stored.

See section 2.1.4 Address modes.

2.6.26. SETx [BBa.r+#S12],BBx.r

Stores a single register to memory using register plus offset address.

See encoding diagram 3.6.26

Syntax

SET[B|W|D] **[Dst],Src**

[Dst]	[BBa.r+#S12]
Src	BBx.r

Operation

```
0:          Addr    = BUAddr(Dst),
            Size    = [B|W|D],
            Data    = Src;
2:          MemStore(Addr, Size, Data);
```

Examples

```
  SETB      [A0FrP+#0x7ff],A0.7
  SETD      [D0Re0+#-0x2000],A1.7
  SETW      [A1LbP+#0xffe],D1Re0
  SETW      [D0Re0+#0xffe],D1.7
```

Usage notes

The same address/data unit may be used to generate the memory address and specify the register to be loaded.

See section 2.1.4 Address modes.

2.6.27. SETLcc [BBa.r],BBx.r,BBo.e

Performs a conditional store of a pair of registers to memory using a register address.

See encoding diagram 3.6.27

Syntax

SETLcc **[Dst],Src1,Src2**

[Dst]	[BBa.r]
Src1	BBx.r
Src2	BBo.e

Operation

```
0:          Addr          = BUAddr(Dst),
            Size          = L,
            Data, Data.o = Src1, Src2;
2: <cc> -> MemStore(Addr, Size, Data);
```

Examples

```
  SETLGE    [A0.7],A0.7,A1.7
  SETLGE    [D0.7],D0.7,D1.7
  SETLGT    [A1.7],D0.7,D1.7
  SETLGT    [D1Re0],A0.7,A1.7
```

Usage notes

The same address/data unit may be used to generate the memory address and specify the register to be loaded. No offset can be applied to the address.

See section 2.1.9 Conditional operations.

See section 2.1.4 Address modes.

2.6.28. SETXcc [BBx.r],Ux.r

Performs a conditional store of a single register to memory using register plus offset address.

See encoding diagram 3.6.28

Syntax

SET[B|W|D] **[Dst],Src**

[Dst]	[BBx.r]
Src	BBx.r | PC.r | CT.r | TR.r

Operation

```
0:          Addr   = BUAddr(Dst),
            Size   = [B|W|D],
            Data   = Src;
2: <cc> -> MemStore(Addr, Size, Data);
```

Examples

```
  SETDCS    [D0.7],D0.7
  SETBPL    [A1.7],A0.7
  SETDMI    [D0.7],PC
  SETDMI    [A1.7],TXSTAT
  SETDLT    [D0.7],TXRPT
  SETDGT    [D1Re0],D0.7
  SETWNV    [A0FrP],D0.7
```

Usage notes

The same address/data unit may be used to generate the memory address and specify the register to be loaded.

PC.r | CT.r | TR.r | TT.r can only be stored from a 32-bit **`[D]`** instruction.

See section 2.1.9 Conditional operations.

2.6.29. SWAPcc Ux.r,Uy.r

Performs a conditional simultaneous transfer between internal processor locations in different units.

See encoding diagram 3.6.29

Syntax

SWAPcc **Dst,Src**

Dst	BBo.r | PC.r | CT.r | TR.r | TT.r
Src	BBx.r | PC.r | CT.r | TR.r | TT.r

Operation

```
0:          XUSrc = Src,
            XUDst = Dst;
1: <cc> ->  'Dst = XUSrc,
   <cc> ->  'Src = XuDst;
```

Examples

```
  SWAPCS    A0.7,D0Re0
  SWAPCC    PC,A1.7
  SWAPHI    TTMARK,D1Re0
  SWAPPL    A0.7,A1.7
  SWAPNV    TXIDLECYC,A1LbP
```

Usage notes

Dst and Src must be in different units.

SWAP PC, Ux.r effectively implements a call 'to address value in register'.

PC.r | CT.r | TR.r | TT.r must be treated as if they are a single unit. So it is not possible to SWAP within this set of units e.g. SWAP CT.r, PC.r is not allowed.

See section 2.1.9 Conditional operations.

2.6.30. XFRX [BBxUaRoPp],[BByUaRoPp]

Performs a pipelined memory transfer operation.

See encoding diagram 3.6.30

Syntax

XFR[D|L] **[Dst],[Src]**

[Dst]	[BBxUaRoPp]
[Src]	[BByUaRoPp]

Operation

```
(n*2)+0:                  DstAddr = BUAddr(Dst),
                          DstSize = [D|L],
                          DstData = RD,
         (TXRPT != 0) -> TXRPT' = TXRPT - 1;
(n*2)+1:                  SrcAddr = BUAddr(Src),
                          SrcSize = [D|L];
(n*2)+2:                  MemStore(DstAddr, DstSize, DstData);
(n*2)+3:                  SrcData = RAMov(SrcAddr, SrcSize);
(n*2)+4:                  'RD     = RAMod(SrcData, RA, SrcSize);
```

Examples

```
  XFRD     [A0.7+A0.7],[A1LbP+A1.7]
  XFRD     [D1.7+D1.7],[A0.7+A0.7]
  XFRL     [D1Re0+D1Re0],[A0.7+A0FrP++]
  XFRL     [A0FrP+A0FrP++],[D1Re0+D1.7]
```

Usage notes

One of `[D]` or `[L]` must be specified.

In order to further optimise larger transfers it is possible to pre-issue up to six addresses before the XFR operation is performed - for each pre-issue a corresponding SETD/SETL operation must be performed after the pipelined transfers are complete. The number of pre/post operations appropriate in any particular case is up to the programmer to decide.

See section 2.1.4 Address modes.

2.6.31. XSDB De.r,Dx.r

Sign extend a byte into a 32-bit result.

See encoding diagram 3.6.31

Syntax

XSD<S>B **Dst,Src**

Dst	De.r
Src	Dx.r

Operation

```
0:          'Dst = DUXsdb(Src);
```

Examples

```
  XSDB      D0Re0,D0.7
  XSDSB     D0Re0,D0.7
```

Usage notes

2.6.32. XSDW De.r,Dx.r

Sign extend a word (16-bit) into a 32-bit result.

See encoding diagram 3.6.32

Syntax

XSD<S>W **Dst,Src**

Dst	De.r
Src	Dx.r

Operation

```
0:          'Dst = DUXsdw(Src);
```

Examples

```
   XSDW     D0Re0,D0.7
   XSDSW    D0Re0,D0.7
```

Usage notes

2.7. Long cache operations

2.7.1. CACHERL BBx.r,BBo.e,[BBa.r+#S6]

Performs specialised 64-bit read to request a linear to physical address translation using the hardware mechanisms provided by the data or instruction caches.

See encoding diagram 3.7.1

Syntax

CACHERL **Dst1,Dst2,[Src]**

Dst1	BBx.r
Dst2	BBo.e
Src	[BBa.r+#S6]

Operation

```
0:     LineAddr          = BUAddr(Src),
       Size              = L;
2:     PhyAddr, TLBFlags = AddrTrans(LineAddr, Size);
3:     'Dst1, 'Dst2      = PhyAddr, TLBFlags;
```

Examples

```
  CACHERL   A0.7,A1.7,[A1.7+#0x60]
  CACHERL   A0FrP,A1LbP,[D0Re0+#0xa0]
  CACHERL   D0Re0,D1Re0,[A0FrP+#-0x38]
  CACHERL   D1.7,D0.7,[D1.7+#0xf8]
```

Usage notes

See section 2.1.4 Address modes.

The type of operation performed is selected by the lowest 6 bits of the source address field. The operations chosen by this are:

Bit 5:4 - Reserved; set to zero

Bit 3 - 0 Data Cache operation

1 Code Cache operation

Bit 2:0 - Reserved; set to zero

The result contains the translated address in the lower 32-bits and the TLB flags and other data in the top 32-bits. Example:

```
AND     D1Ar1, D1Ar1, #(-1<<6)        ; Clear bits 5-0 of address
CACHERL D0Re0, D1Re0, [D1Ar1+#8]      ; Code cache TLB data look-up
```

See section on Instruction and data caches in the Architecture Overview TRM for details of the various fields of the returned data.

2.7.2. CACHERD BBx.r,[BBa.r+#S6]

Performs specialised 32-bit read to request a linear to physical address translation using the hardware mechanisms provided by the data or instruction caches.

See encoding diagram 3.7.2

Syntax

CACHERD **Dst,[Src]**

Dst	BBx.r
Src	[BBa.r+#S6]

Operation

```
0:     LineAddr = BUAddr(Src),
       Size     = D;
2:     PhyAddr  = AddrTrans(LineAddr, Size);
3:     'Dst     = PhyAddr;
```

Examples

```
  CACHERD   A1.7,[D1.7+#-0x5c]
  CACHERD   D1Re0,[D1Re0+#0x28]
```

Usage notes

See section 2.1.4 Address modes.

The type of operation performed is selected by the lowest 6 bits of the source address field. The operations chosen by this are:

Bit 5:4 - Reserved; set to zero

Bit 3 - 0 Data Cache operation

1 Code Cache operation

Bit 2:0 - Reserved; set to zero

The result contains the translated address.

```
AND     D1Ar1, D1Ar1, #(-1<<6)        ; Clear bits 5-0 of address
CACHERD D0Re0, [D1Ar1]                ; Data cache TLB data look-up
```

See section on Instruction and data caches in the Architecture Overview TRM for details of the various fields of the returned data.

2.7.3. CACHEWL [BBa.r+#S6],BBx.r,BBo.e

Performs a specialised 64-bit write to the data cache that is used to instruct the data and instruction cache to perform a data cache or instruction cache linear flush,
or a data cache or instruction cache TLB linear flush.

See encoding diagram 3.7.3

Syntax

CACHEWL **[Dst],Src1,Src2**

[Dst]	[BBa.r+#S6]
Src1	BBx.r
Src2	BBo.e

Operation

```
0:     Addr   = BUAddr(Dst),
       Size   = L,
       Data   = Src1, Src2;
2:     CacheFlush(Addr, Size, Data);
```

Examples

```
  CACHEWL   [A0.7+#-0x180],D1Re0,D0Re0
  CACHEWL   [A1.7+#0x0],A0.7,A1.7
  CACHEWL   [D1Re0+#0x740],A0.7,A1.7
  CACHEWL   [D1Re0+#0x240],D1.7,D0.7
```

Usage notes

See section 2.1.4 Address modes; note that the immediate 6-bit offset supported is scaled by the size of a cache line (64-bytes) not units of 64-bits as for other 64-bit (L) memory operations.

The type of operation performed is selected by the data presented to the data cache. Only the bottom eight bits of data are used. The operations chosen by this data are:

Bit 0 - Data cache (0) or instruction cache (1) operation.

Bit 1 - Cache ways linear flush (0) or TLB linear flush (1) operation.

Bit 2-7 - Reserved.

2.7.4. CACHEWD [BBa.r+#S6],BBx.r

Performs a specialised 32-bit write to the data cache that is used to instruct the data and instruction cache to perform a data cache or instruction cache linear flush,
or a data cache or instruction cache TLB linear flush.

See encoding diagram 3.7.4

Syntax

CACHEWD **[Dst],Src**

[Dst]	[BBa.r+#S6]
Src	BBx.r

Operation

```
0:     Addr    = BUAddr(Dst),
       Size    = D,
       Data    = Src;
2:     CacheFlush(Addr, Size, Data);
```

Examples

```
  CACHEWD   [A0.7+#0x240],A0FrP
  CACHEWD   [A0FrP+#0x580],D0.7
  CACHEWD   [D1.7+#-0x40],A1.7
  CACHEWD   [D1Re0+#0x80],D0.7
```

Usage notes

See section 2.1.4 Address modes; note that the immediate 6-bit offset supported is scaled by the size of a cache line (64-bytes) not units of 32-bits as for other 32-bit (D) memory operations.

The type of operations performed is selected by the data presented to the data cache. Only the bottom eight bits of data are used. The operations chosen by this data are:

Bit 0 - Data cache (0) or instruction cache (1) operation.

Bit 1 - Cache ways linear flush (0) or TLB linear flush (1) operation.

Bit 2-7 - Reserved.

2.7.5. DCACHE [BBa.r+#S6],BBx.r

Performs a specialised write to the data cache that is used to instruct the cache to take a specified action for any data cached on the same line as the write address.

See encoding diagram 3.7.5

Syntax

DCACHE **[Dst],Src**

[Dst]	[BBa.r+#S6]
Src	BBx.r

Operation

```
0:    Addr   = BUAddr(Dst),
      Data   = Src;
2:    DCacheLineFlush(Addr, Data);
```

Examples

```
  DCACHE    [A1.7+#-0x200],A0.7
  DCACHE    [A1LbP+#-0x780],D1Re0
  DCACHE    [D1Re0+#0x780],A1.7
  DCACHE    [D1Re0+#0x7c0],D1Re0
```

Usage notes

See section 2.1.4 Address modes; note that the immediate 6-bit offset supported is scaled by the size of a cache line (64-bytes) not units of 32-bits as for other 32-bit (D) memory operations.

For this instruction a memory write is performed with additional sideband set to indicate that the write is for a flush. The data associated with the memory write may be used to control the action of the cache in response to the linear flush command ('linear' as the addresses used are all pre-cache as opposed to post-cache and MMU).

The content of the Src register is reserved and must always be set to zero.

2.7.6. ICACHE #S15,#X4

This cache prefetch instruction allows software to cause the instruction cache to be preloaded with several cache lines of data starting from a software defined position (e.g. function name or code label).

See encoding diagram 3.7.6

Syntax

ICACHE[R] **Src1,Src2**

Src1	#S15 | Label
Src2	#X4 (Number of 64-byte cache lines) | Label

Operation

```
0:          Addr = PCUAddr(Src1),
            Count = Src2,
            Pri = R;
2:          ICACHEPrefetch(Addr, Count, Pri);
```

Examples

```
  ICACHE    #+16383,#0x4
  ICACHE    #-1,#0x6
  ICACHE    Label,#0x1
```

Usage notes

Setting the [R] flag gives the prefetch operations priority over normal cache fills triggered by misses.

The icache instruction takes a 15-bit signed offset (-16384 <= Value < 16384) in terms of 64 byte units (i.e. the offset undergoes a fixed multiply by 64 before use), which gives a complete offset range of -1048576 to 1048512 bytes from the base of the icache instruction.

The number of cache lines counts from, and includes, the line containing the resulting offset address.

Constant offsets specify number of cache lines, relative to the next cache line.

See next ICACHEEXIT for usage.

2.7.7. ICACHEEXIT

This cache prefetch instruction allows software to terminate a pending intruction cache prefetch operation started via a preceding ICACHE instruction.

See encoding diagram 3.7.7

Syntax

ICACHEEXIT[R]

Operation

```
0:          Addr = PCUAddr(0),
            Count = 0,
            Pri = R,
2:          ICACHEPrefetch(Addr, Count, Pri);
```

Examples

```
ICACHEEXIT
```

Usage notes

The [R] flag should be (by convention) set if [R] was set on the corresponding ICACHE operation.

This operation is typically used to ensure that any pending ICACHE operation is terminated prior to leaving the function that would benefit from it.

```
        ICACHE Label, #6        ; Load 6 cache lines of code into instruction cache
         ...
Label:  ...                     ; Code being preloaded
         ...
FnExit: ICACHEEXIT              ; Stop and pending ICACHE operation before returning to caller
         MOV PC, D1RtP          ; Return
```

2.8. Long trace operations

2.8.1. MOV TT.r,#I16

Primes a real-time trace unit register with 16-bit data.
See encoding diagram 3.8.1

Syntax

MOV	**Dst,Src**
Dst	TT.r
Src	#I16

Operation

```
0:          'Dst    = TTUMov(Src);
```

Examples

```
  MOV       TTMARK,#0x8000
```

Usage notes

2.8.2. TTMOV Ux.r,Uy.r

Performs a cross-unit MOV. In addition to doing the MOV, this instruction sends the source data to the thread's TTREC register in the real-time trace unit.

See encoding diagram 3.8.2

Syntax

TTMOV **Dst,Src**

Dst	Dx.r | Ax.r | PC.r | CT.r | TR.r | TT.r
Src	Dx.r | Ax.r | PC.r | CT.r | TR.r | TT.r

Operation

```
0:          XUVal  = XUMov(Src);
1:          'Dst   = XUVal,
            'TTREC = XUVal;
```

Examples

```
  TTMOV     A0.3, A1.0        ; Unit-to-unit move with Src output to TTREC
  TTMOV     D0FrT,A0FrP       ; Trace push of frame pointer
  TTMOV     A0FrP,D0FrT       ; Trace pop of frame pointer
```

Usage notes

This instruction is used to replace a similar normal MOV operation so that the data transferred can be recorded by the trace unit without a net increase in the size of the code concerned. This is typically used for marking function entry and exit sequences that contain such cross-unit MOV operations.

2.8.3. MOVLcc TTREC,BBx.r,BBo.r

Performs a 64-bit wide transfer from a pair of address or data unit registers to a thread's TTREC register in the real-time trace unit.

See encoding diagram 3.8.3

Syntax

MOVLcc **Dst,Src1,Src2**

Dst	TTREC
Src1	Dx.r | Ax.r|CPCx
Src2	Do.r | Ao.r|CPCe

Operation

```
0:          XUVal, XUVal.o = Src1, Src2;
1:  <cc> -> 'Dst, 'Dst.o = XUVal, XUVal.o;
```

Examples

```
  MOVLCS    TTREC,D1.7,D0.7
  MOVLLT    TTREC,A0.7,A1.7
```

Usage notes

Src1 and Src2 must be in opposite units of the same type.

The low 32-bits of the data come from Src1 while the remaining 32-bits come from Src2.

See section 2.1.9 Conditional operations.

3. Operation coding diagrams

This section gives detailed implementation notes describing how the GP instructions are encoded.

3.1. Encoding notation

3.1.1. Op-code fields

The op-code field is eight bits in length. In this field four bits define the primary operation type, these are defined as:

Op Type	Code	Mnemonic
Data Unit Ops	0	ADD/MOV
	1	SUB/NEG
	2	AND/ADDB8/ADDT8
	3	OR/SUBB8/SUBT8
	4	XOR/MULB8/MULT8
Data Unit Shift Ops	5	LSL/LSR/ASL/ASR
Data Unit Multiply Ops	6	MUL
Data Unit Comparison	7	CMP/TST/FFB/NORM/MIN/MAX/ABS
Address Unit Ops	8	ADD/SUB/MOV/NEG
U2UMisc	A	B<cc>/MOV (port-to-unit, unit-to-unit, etc.)
Memory Ops	B	SET (to mem)
	C	GET (from mem)/MDRD/MMOV
	D	XFR (mem to mem)
Coprocessor Ops	E	CPW

A further four bits, defined as the secondary op-code, are decoded as parameter options on the above basic instructions. These bits have the following meanings:

Parameter	Meaning
I	Use immediate data
AU	Addr unit select ('0'=A0, '1'=A1)
DU	Data unit select ('0'=D0, '1'=D1)
S	Set Condition Flags (from DU only)
C	Conditional
T	Select SUB (T='0') or AND (T='1') for comparison i.e. CMP/TST
PM	Address unit plus/minus. To ADD PM must be' 0', while '1' will perform a SUB.
SW	Swap Us.Rs with Ud.Rd (unit-to-unit ops),
DS	Coprocessor data source ('0'=Register, '1'=Memory)

Parameter	Meaning
L2/L1	Transfer Width specifier (00=8-bit, 01=16-bit, 10=32-bit, 11=64-bit). When there is no L2 flag, L1 selects between 32 and 64-bit.
L2	In the case of MUL, L2=0 is a 16-bit unsigned multiply, L2=1 is a 32-bit unsigned multiply.

In addition the following bits may reside in the parameter field (bits 0-23):

Parameter	Meaning
O1Z	Operand 1 zero ('0'=>op1=Rd/Rs1, '1'=>op1=ZERO)
H	Load 16-bit immediate into high 16-bits (H and SE cannot be set together)
CA	Condition is Always
SE	Sign Extend 16-bit immediate (H and SE cannot be set together)
AS	Arithmetic shift ('0'=LSx, '1'=ASx)
SR	Shift right ('0'=xSL, '1'=xSR)
SU	Signed/Unsigned multiply ('0'=unsigned, '1'=signed)
LF	Lock Flag ('0'=release lock, '1'=acquire lock)
LA	Lock All ('0'=release lock, '1'=acquire lock)
UA	Update Address inc UD, US variants ('1'=update, '0'=don't update)
BU	Base Unit inc BUS, BUD, BUA variants (01=D0, 10=D1, 11=A0 and 00=A1)
PP	Pre/Post inc ('0'=pre, '1'=post, ignored if not updating the address)
M	Make an AND mask (by padding 32-bit word made from 16-bit immediate with '1's)
IO	Read address retrieval (IO='0') or reissue (IO='1')
R	Repeat
P	Priority (see ICACHE and ICACHEEXIT)
PC	Address unit PC relative addressing ('1' indicates PC.0 will be used for operand).

3.1.2. Condition codes

Encoding	Condition	Meaning	Condition 'Flag' State
0000	A	Always	
0001	EQ/Z	Equal/Zero	Zero == 1
0010	NE/NZ	Not Equal/Not Zero	Zero == 0
0011	CS/LO	Carry Set/Lower	**C**arry == 1
0100	CC/HS	Carry Clear/Higher or Same	**C**arry == 0
0101	MI/N	Minus/N Set	Neg == 1

Encoding	Condition	Meaning	Condition 'Flag' State
0110	PL/NC	Positive/N Clear	Neg == 0
0111	VS	Overflow Set	**O**ver == 1
1000	VC	Overflow Clear	**O**ver == 0
1001	HI	Higher (unsigned)	(**C**arry == 0) AND (**Z**ero == 0)
1010	LS	Lower or Same (unsigned)	(**C**arry == 1) OR (**Z**ero == 1)
1011	GE	Greater Than or Equal	**N**eg == **O**ver
1100	LT	Less Than	**N**eg != **O**ver
1101	GT	Greater Than	(**Z**ero == 0) AND (**N**eg == **O**ver)
1110	LE	Less Than or Equal	(**Z**ero == 1) OR (**N**eg != **O**ver)
1111	NV	Never	

3.1.3. Register field encoding

In the majority of Meta 32-bit instructions and some MiniM instructions, registers are encoded in a 5-bit field according to the table below. See the Architecture Overview TRM for details of control registers.

Meta 5-bit register encodings

Reg		**BB (Us/Ud 1:0)**							
	CT	D0	D1	A0	A1	PC	Ports	TR	TT
	0	1	2	3	4	5	6	7	8
0	TXENABLE	D0Re0	D1Re0	A0StP	A1GbP	PC	CP.0	TXSTAT	TTEXEC
1	TXMODE	D0Ar6	D1Ar5	A0FrP	A1LbP	PCX	CP.1	TXMASK	TTCTRL
2	TXSTATUS	D0Ar4	D1Ar3	A0.2	A1.2		CP.2	TXSTATI	TTMARK
3	TXRPT	D0Ar2	D1Ar1	A0.3	A1.3		CP.3	TXMASKI	TTREC
4	TXTIMER	D0FrT	D1RtP	A0.4	A1.4		CP.4	TXPOLL	GTEXEC
5		D0.5	D1.5	A0.5	A1.5		CP.5		
6		D0.6	D1.6	A0.6	A1.6		CP.6	TXPOLLI	
7		D0.7	D1.7	A0.7	A1.7		CP.7		
8		D0.8	D1.8	GblA0.8	GblA1.8				
9		D0.9	D1.9	GblA0.9	GblA1.9				
10		D0.10	D1.10	GblA0.10	GblA1.10				
11	TXBPOBITS	D0.11	D1.11	GblA0.11	GblA1.11				
12		D0.12	D1.12	GblA0.12	GblA1.12				

Reg		BB (Us/Ud 1:0)							
13	TXTIMERI	D0.13	D1.13	GblA0.13	GblA1.13				
14		D0.14	D1.14	GblA0.14	GblA1.14				
15		D0.15	D1.15	GblA0.15	GblA1.15				
16	TXCATCH0	GblD0.16	GblD1.16	CPC0	CPC1		RA | RD		
17	TXCATCH1	GblD0.17	GblD1.17				RAPF		
18	TXCATCH2	GblD0.18	GblD1.18						
19	TXCATCH3	GblD0.19	GblD1.19						
20	TXDEFR	GblD0.20	GblD1.20						
21	Reserved	GblD0.21	GblD1.21						
22	TXCLKCTRL	GblD0.22	GblD1.22				RAM8X32		
23	TXSTATE	GblD0.23	GblD1.23				RAM8X		
24	TXAMAREG0	GblD0.24	GblD1.24				RABZ		
25	TXAMAREG1	GblD0.25	GblD1.25				RAWZ		
26	TXAMAREG2	GblD0.26	GblD1.26				RADZ		
27	TXAMAREG3	GblD0.27	GblD1.27						
28	TXDIVTIME	GblD0.28	GblD1.28				RABX		
29	TXPRIVEXT	GblD0.29	GblD1.29				RAWX		
30	TXACTCYC	GblD0.30	GblD1.30				RADX		
31	TIDLECYC	GblD0.31	GblD1.31				RAM16X		

Note: Registers Dx.8 to Dx.15 and Ax.4 to Ax.7 are only present on DSP enabled cores, but when present they are available to both GP and DSP instructions.

MiniM 3-bit register encodings

In most MiniM instructions the register field is encoded in a 3-bit field as follows:

R			BB (Us/Ud 1:0)								
	CT_X	CT	D0	D1	A0	A1	PC	Raxx	Ports	TR	TT
	F	0	1	2	3	4	5	E	6	7	8
0	TXCATCH0	TXENABLE	D0Re0	D1Re0	A0StP	A1GbP	PC	RABZ	CP.0	TXSTAT	TTEXEC
1	TXCATCH1	TXMODE	D0Ar6	D1Ar5	A0FrP	A1LbP	PCX	RAWZ	CP.1	TXMASK	TTCTRL
2	TXCATCH2	TXSTATUS	D0Ar4	D1Ar3	A0.2	A1.2		RADZ	CP.2	TXSTATI	TTMARK
3	TXCATCH3	TXRPT	D0Ar2	D1Ar1	A0.3	A1.3			CP.3	TXMASKI	TTREC
4	TXDEFR	TXTIMER	D0FrT	D1RtP				RABX	RA | RD	TXPOLL	GTEXEC
5	TXTIMERI		D0.5	D1.5				RAWX	RAPF		
6	TXACTCYC		D0.6	D1.6				RADX	RAM8X32	TXPOLLI	
7	TIDLECYC		D0.7	D1.7				RAM16X	RAM8X		

The Reg, or R, column shows the decimal value of the 3-bit or 5-bit Rs/Rd field which is coded least significant bit first. In address unit operations the register encoding uses the first four bits of the 5-bit encoding, the 5th bit being the PC bit. The execution unit (CT, D0, D1 etc.) is inferred either from the opcode group as shown in section 3.1.1. or from other fields in the instruction encoding, namely the BU field (01=D0, 10=D1, 11=A0, 00=A1) and the explicit instruction specifiers Us/Ud shown below in the table Explicit unit specifiers.

Split MiniM register encodings

In MiniM extended cases where the register encoding is split across two instruction words, the fields are combined so that the 3-bit field becomes the first word and the 2-bit field becomes the second word of the standard Meta 5-bit register encoding. Where there is no field defined for the second word in the encoding diagram, the register definition is truncated to a 3-bit field. This restricts the available registers to the standard GP subset of the 5-bit encodings, the higher order registers, Global and DSP, being unavailable for these operations.

Explicit unit specifiers

In some instructions the execution unit (source or destination) is specified in explicit form. In these cases the following 4-bit patterns apply:

Unit No	Encoding	**Source Unit** (Us)	**Destination Unit** (Ud)
0	0000	CT Internal Control Registers	CT Internal Control Registers
1	0001	D0	D0
2	0010	D1	D1
3	0011	A0	A0
4	0100	A1	A1
5	0101	PC	PC
6 *	0110	Ports (RD*/CPRD)	Ports (RA*)
7	0111	Triggers	Triggers
8	1000	TT Trace Registers	TT Trace Registers
9	1001	FX	FX
E **	1110	RAxx Read Address Ports	
F **	1111	CT_X Internal Control Registers	CT_X Internal Control Registers

Notes:

** The register encodings in this unit are different between Meta and MiniM instructions.*

*** Only present for MiniM instruction.*

Operand 2 replace (O2R)

A number of instructions that use two register operand fields support O2R.This flag indicates that the second register operand will be replaced with data sourced from either the memory read data port or from a register in one of the other address or data units.

When O2R is used the Rs2 register field will be decoded in a special way with the top two bits of the field identifying the source unit for the replacement data and the bottom three bits indicating a register number (where appropriate). The actual encoding of the source unit options varies depending upon which unit is being used as the source for register operand 1. In some address unit instructions the Rs field is 4 bit with the PC bit where the 5th bit would be. When O2R is set however the PC bit is treated as part of the Rs2 field as far as O2R is concerned.

For data unit operations the encoding for instructions where DU is '0' (i.e. D0 is used) is shown below:

Source Unit Bits	Replacement Source Unit
00	A1.r
01	D1.r
10	RD
11	A0.r

For data unit operations the encoding for instructions where DU is '1' (i.e. D1 is used) is shown below:

Source Unit Bits	Replacement Source Unit
00	A1.r
01	D0.r
10	RD
11	A0.r

For address unit operations the encoding for instructions where AU is '0' (i.e. A0 is used) is shown below:

Source Unit Bits	Replacement Source Unit
00	A1.r
01	D0.r
10	RD
11	D1.r

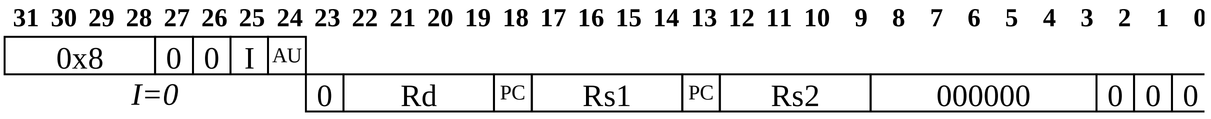
For address unit operations the encoding for instructions where AU is '1' (i.e. A1 is used) is shown below:

Source Unit Bits	Replacement Source Unit
00	D1.r
01	D0.r
10	RD
11	A0.r

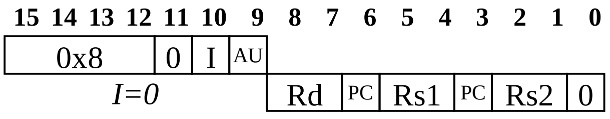
3.2. Core general purpose instructions

These instructions all have a short encoding format in common usage.

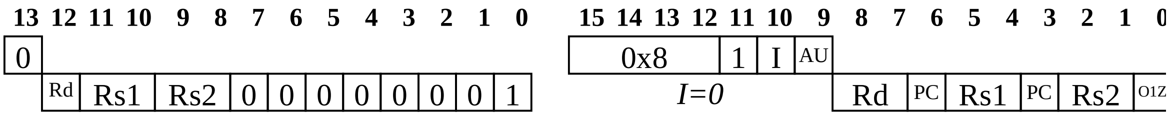
3.2.1. ADD Ae.r,Ax.r,Ae.r



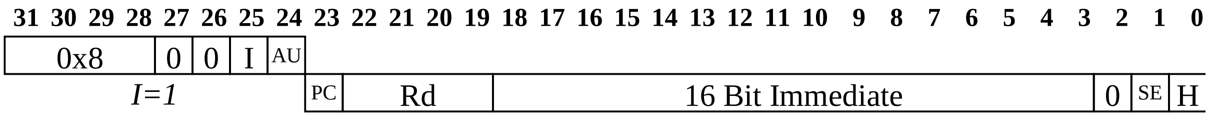
MiniM core encoding



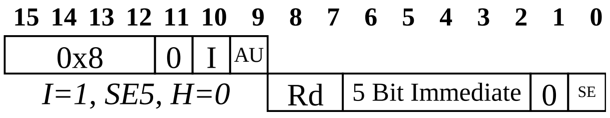
MiniM extended encoding



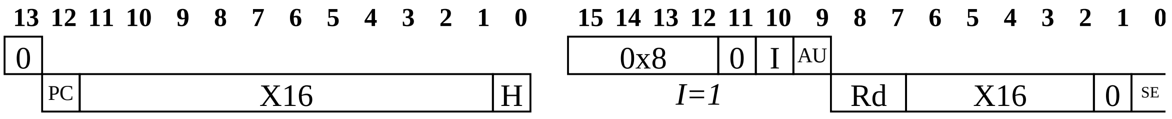
3.2.2. ADD Ae.r,Ax.r,#I16



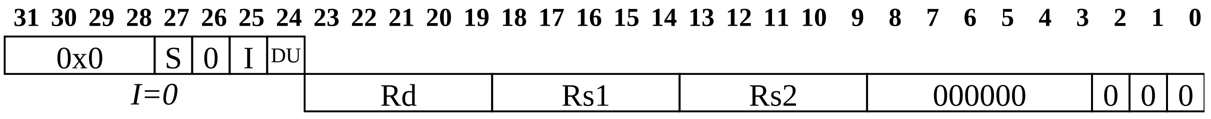
MiniM core encoding



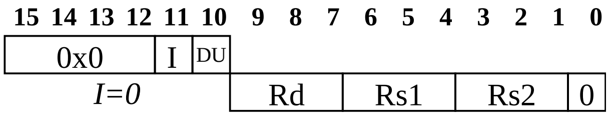
MiniM extended encoding



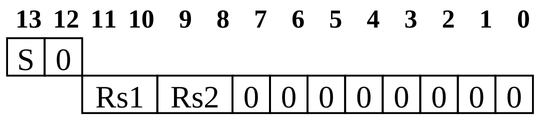
3.2.3. ADD De.r,Dx.r,De.r

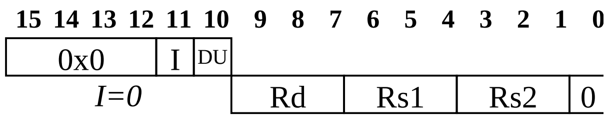


MiniM core encoding

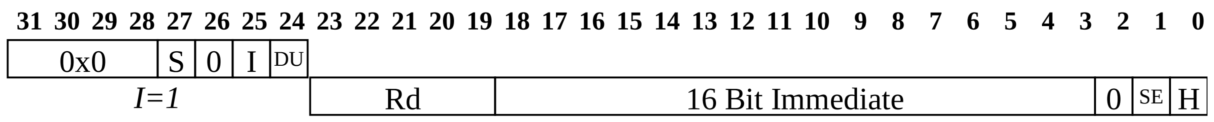


MiniM extended encoding

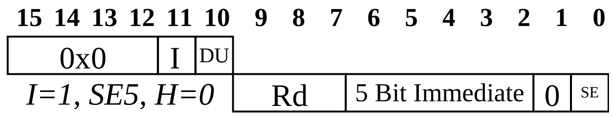




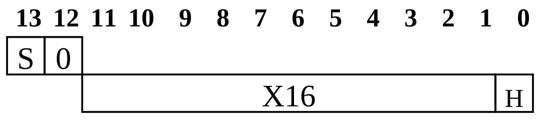
3.2.4. ADD De.e,Dx.r,#I16

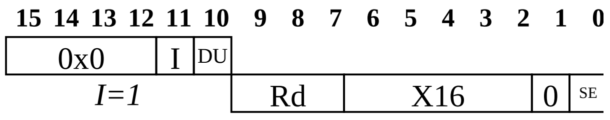


MiniM core encoding

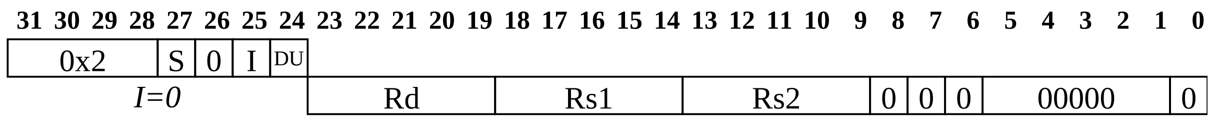


MiniM extended encoding

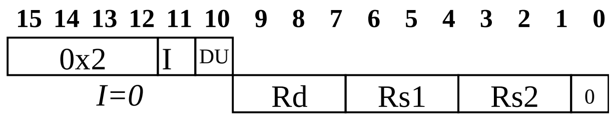




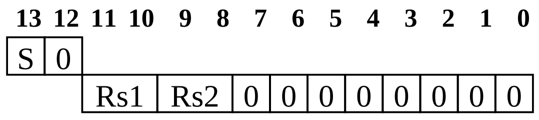
3.2.5. AND De.r,Dx.r,De.r

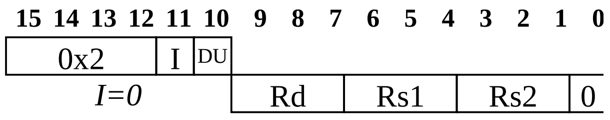


MiniM core encoding

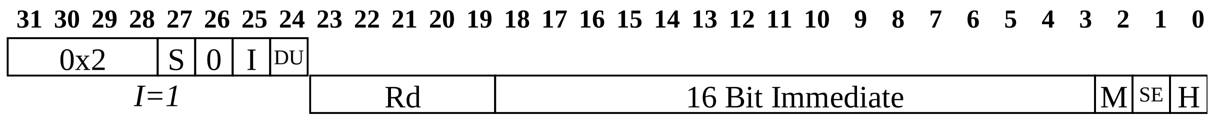


MiniM extended encoding

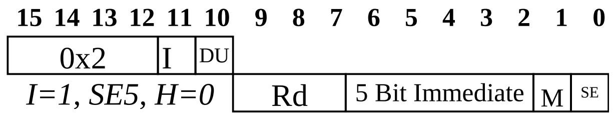




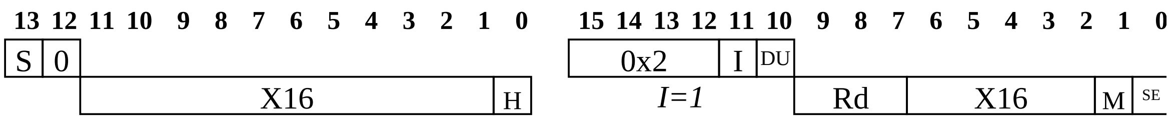
3.2.6. AND De.e,Dx.r,#I16



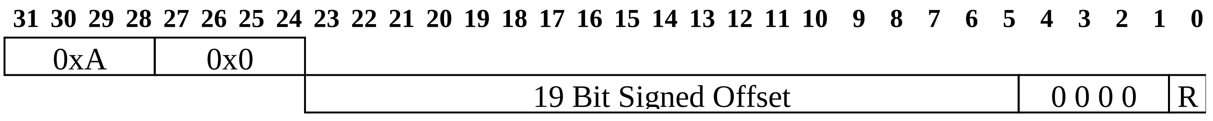
MiniM core encoding



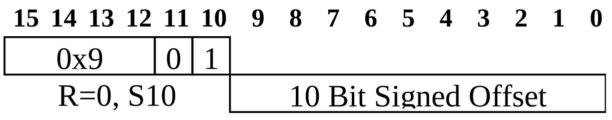
MiniM extended encoding



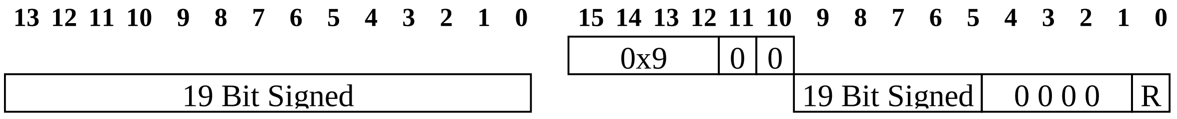
3.2.7. B #S19



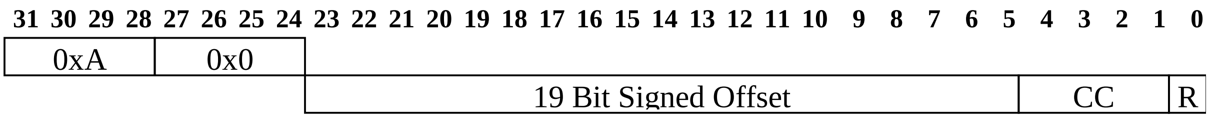
MiniM core encoding



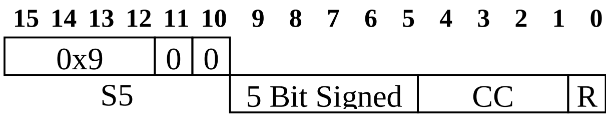
MiniM extended encoding



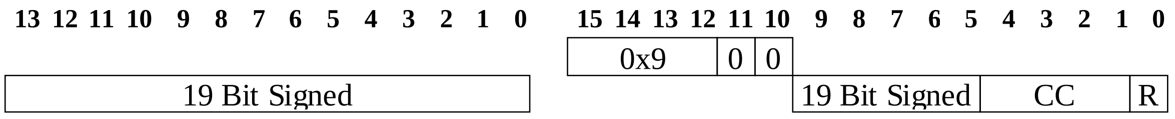
3.2.8. Bcc #S19



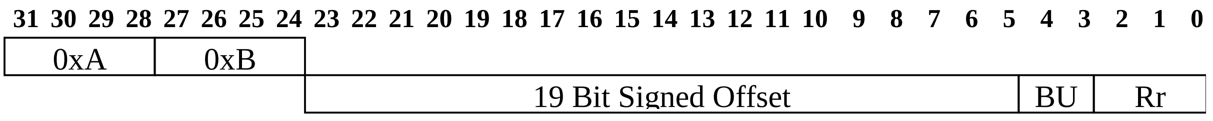
MiniM core encoding



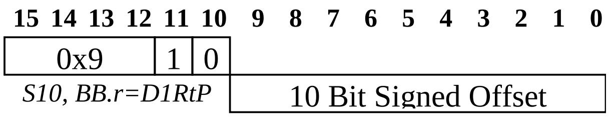
MiniM extended encoding



3.2.9. CALLR BBx.r,#S19



MiniM core encoding



MiniM extended encoding

13	12	11	10	9	8	7	6	5	4	3	2	1	0
19 Bit Signed									BU		Rr		

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x9				1	0										
						19 Bit Signed									

3.2.10. CMP Dx.r,De.r

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x7				0	0	I	DU																								
			I=0					00000					Rs1					Rs2					00000000								0

MiniM core encoding

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x7				0	I	DU									
			I=0				Rs1			Rs2					0

MiniM extended encoding

13	12	11	10	9	8	7	6	5	4	3	2	1	0
0													
	0	0	0	0	Rs1		0	0	0	0	0	0	0

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x7				0	I	DU									
			I=0				Rs1			Rs2					0

3.2.11. CMP Dx.r,#I16

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x7				0	0	I	DU																								
			I=1					Rs					16 Bit Immediate																M	SE	H

MiniM core encoding

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x7				0	I	DU									
I=1,SE5,M=0,H=0							Rs			5 Bit					SE

MiniM extended encoding

13	12	11	10	9	8	7	6	5	4	3	2	1	0
0													
	X16											M	H

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x7				0	I	DU									
			I=1				Rs1			X16					SE

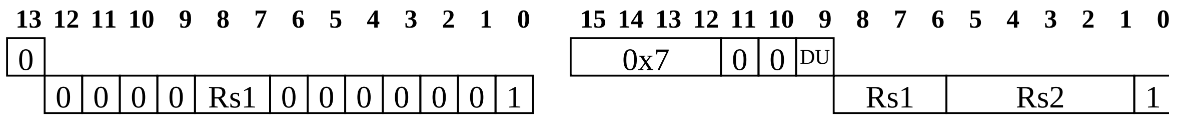
3.2.12. CMP Dx.r,Rx.r

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x7				0	0	0	DU																								
								00000					Rs1					Rs2					00000000								1

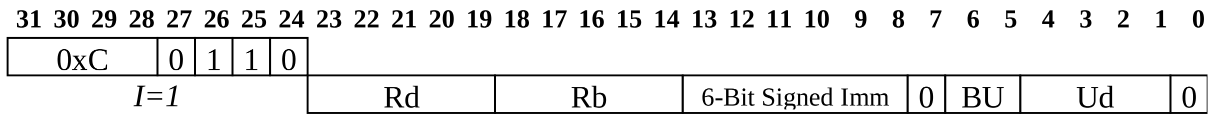
MiniM core encoding

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x7				0	0	DU									
							Rs1			Rs2					1

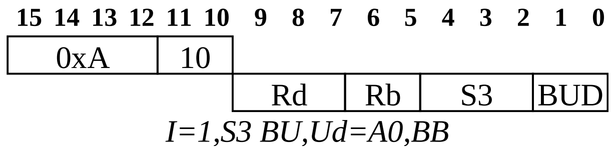
MiniM extended encoding



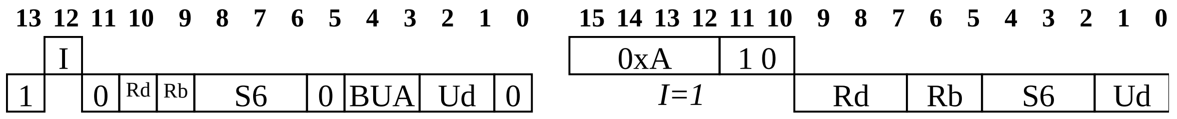
3.2.13. GETD BBx.r, [A0.r+#S3]



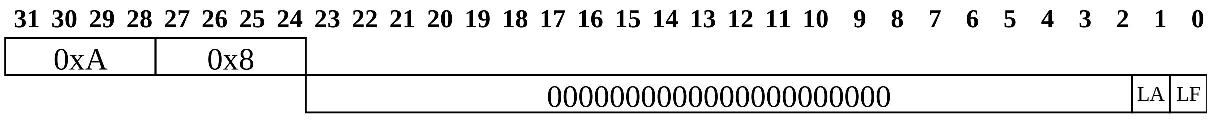
MiniM core encoding



MiniM extended encoding

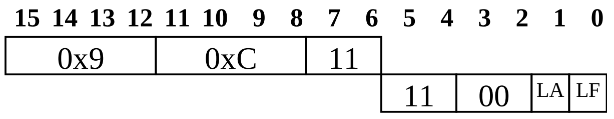


3.2.14. LOCKn

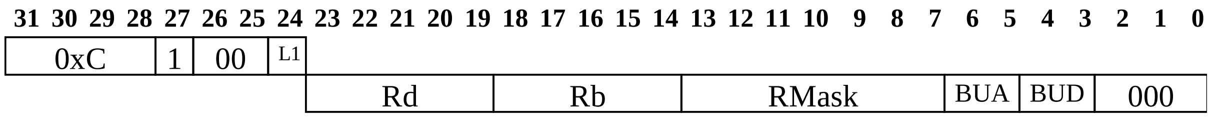


Note: LOCK0: LA=0, LF=0 LOCK1: LA=0, LF=1 LOCK2: LA=1, LF=1

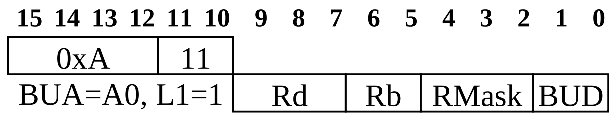
MiniM encoding



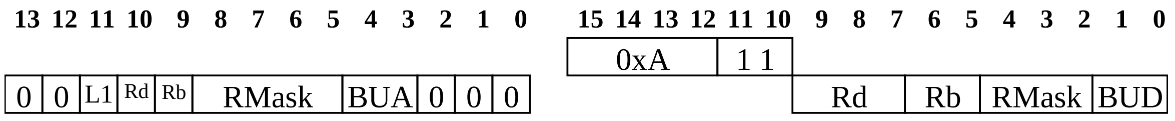
3.2.15. MGET BBx.r..BBe.r..,[BBa.r]



MiniM core encoding



MiniM extended encoding



3.2.16. MOV Ae.r,Ax.r

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x8				0	0	I	AU																								
			I=0					0	Rd				0 0 0 0 0					PC	Rs2				000000						1	0	0

MiniM core encoding

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x8				0	I	AU									
		I=0					Rd		0 0 0			PC	Rs2		1

MiniM extended encoding

13	12	11	10	9	8	7	6	5	4	3	2	1	0
0													
	Rd	0	0	Rs2		0	0	0	0	0	0	0	1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x8				0	I	AU									
		I=0					Rd		0	0	0	PC	Rs2		1

3.2.17. MOV Ax.r,#I16

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x8				0	0	I	AU																								
			I=1					PC	Rd				16 Bit Immediate																1	SE	H

MiniM core encoding

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x8				0	I	AU									
	I=1, SE5, H=0						Rd		5 Bit Immediate					1	SE

MiniM extended encoding

13	12	11	10	9	8	7	6	5	4	3	2	1	0
0													
	PC	X16											H

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x8				0	I	AU									
		I=1					Rd		X16					1	SE

3.2.18. MOV De.r,Dx.r

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x0				S	0	I	DU																								
			I=0					Rd					0 0 0 0 0					Rs2					000000						1	0	0

MiniM core encoding

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x0				I	DU										
		I=0				Rd			0 0 0			Rs2			1

MiniM extended encoding

13	12	11	10	9	8	7	6	5	4	3	2	1	0
S	0												
		0	0	Rs2		0	0	0	0	0	0	0	0

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x0				I	DU										
		I=0				Rd			000			Rs2			1

3.2.19. MOV Dx.r,#I16

31–28	27	26	25	24	23–19	18–3	2	1	0
0x0	S	0	I	DU					
I=1					Rd	16 Bit Immediate	1	SE	H

MiniM core encoding

15–12	11	10	9–7	6–2	1	0
0x0	I	DU				
I=1			Rd	5 Bit Immediate	1	SE

MiniM extended encoding

13	12	11–1	0
S	0		
		X16	H

15–12	11	10	9–7	6–2	1	0
0x0	I	DU				
I=1			Rd	X16	1	SE

3.2.20. MOV Ux.r,Uy.r

31–28	27–24	23–19	18–14	13–10	9	8–5	4–1	0
0xA	0x3							
		Rs	Rd	Us	0	Ud	0 0 0 0	0

MiniM core encoding

15–12	11–8	7	6–4	3–2	1–0
0x9	0xD				
Ud=Ao, Us=Ax		AU	Rs	Rd	BUS

15–12	11–8	7	6	5–3	2–0
0x9	0xE				
Ud=Do, Us=Dx		Ud	Us	Rs	Rd

15–12	11–8	7	6–5	4–2	1–0
0x9	0xD				
Ud=Dx, Us=Ax		DU	Rs	Rd	BUS

15–12	11–8	7	6	5–4	3–2	1	0
0x9	0xE						
Ud=Ax, Us=Dx		0	0	Rs	Rd	Ud	Us

15–12	11–8	7–6	5–3	2	1–0
0x9	0xC	00			
Ud=PC, Us=Dx|Ax			Rs	Rd	BUS

MiniM extended encoding

13–11	10–8	7–4	3–0
Rs	Rd	Us	Ud

15–12	11–8	7–4	3–2	1–0
0x9	0xC	0x8		
			Rs	Rd

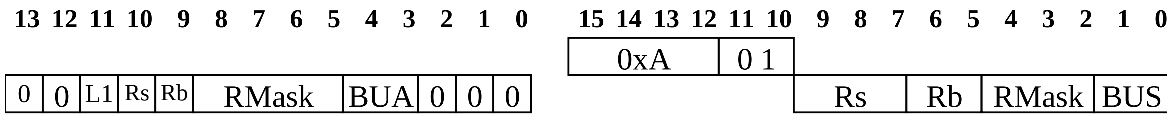
3.2.21. MSETx [BBa.r],BBx.r..BBe.r..

31–28	27	26–25	24	23–19	18–14	13–7	6–5	4–3	2–0
0xB	1	00	L1						
				Rs	Rb	RMask	BUA	BUS	000

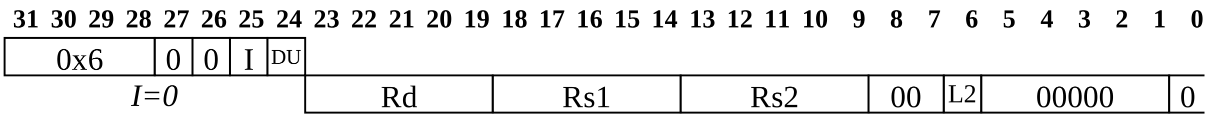
MiniM core encoding

15–12	11–10	9–7	6–5	4–2	1–0
0xA	01				
BUA=A0, L1=1		Rs	Rb	RMask	BUS

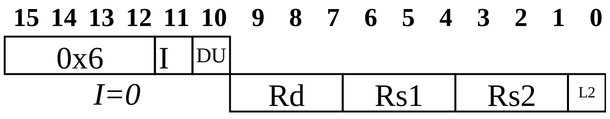
MiniM extended encoding



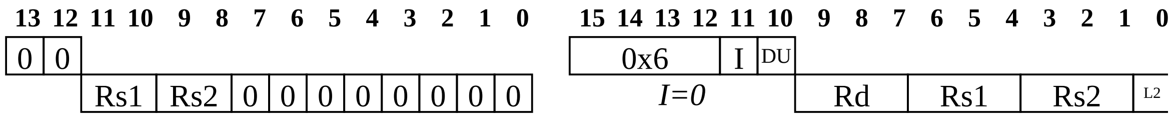
3.2.22. MUL De.r,Dx.r,De.r



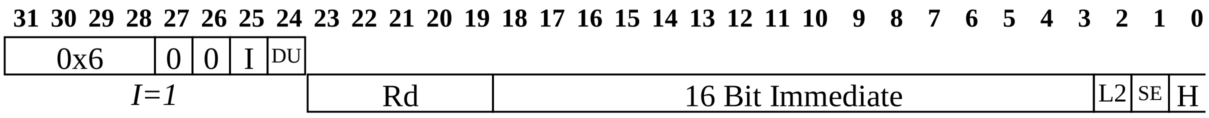
MiniM core encoding



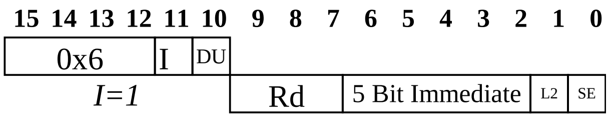
MiniM extended encoding



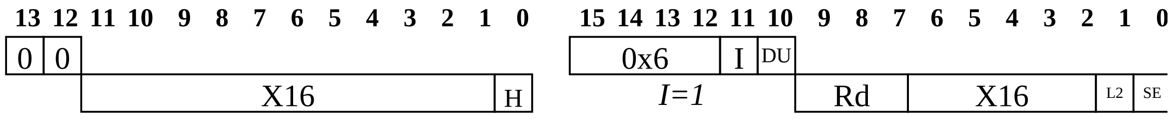
3.2.23. MUL De.r,Dx.r,#I16



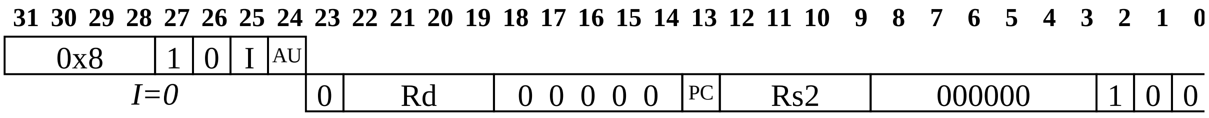
MiniM core encoding



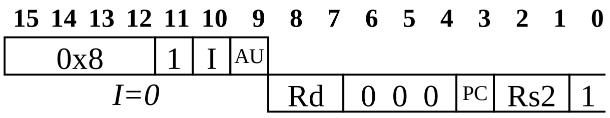
MiniM extended encoding



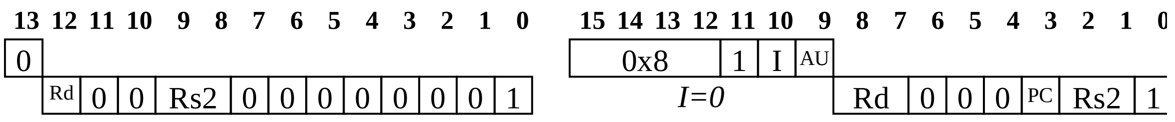
3.2.24. NEG Ae.r,Ax.r



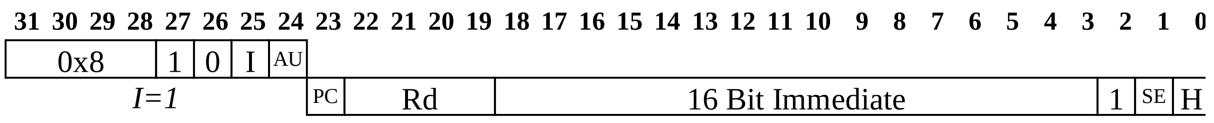
MiniM core encoding



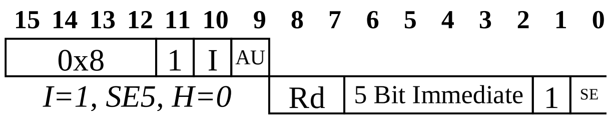
MiniM extended encoding



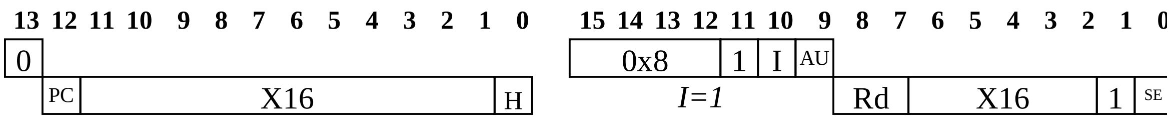
3.2.25. NEG Ax.r,#I16



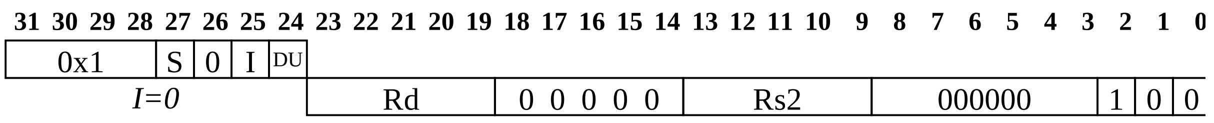
MiniM core encoding



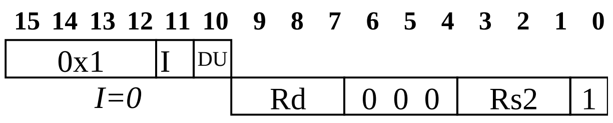
MiniM extended encoding



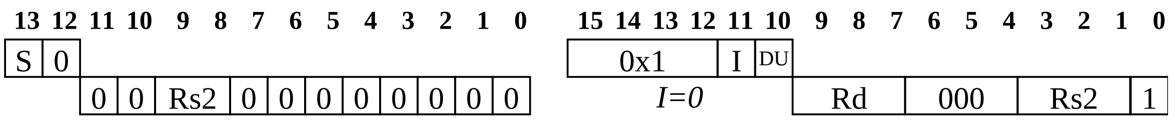
3.2.26. NEG De.r,Dx.r



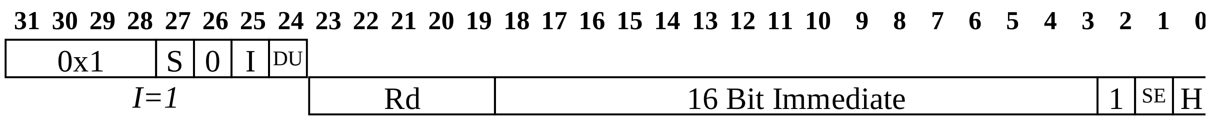
MiniM core encoding



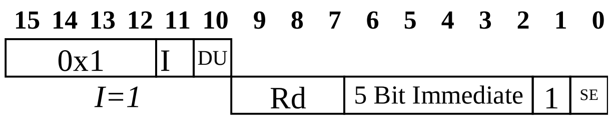
MiniM extended encoding



3.2.27. NEG De.r,#I16



MiniM core encoding



MiniM extended encoding

13	12	11	10	9	8	7	6	5	4	3	2	1	0
S	0												
		X16											H

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x1				I	DU										
I=1						Rd			X16					1	SE

3.2.28. NOP

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0xA				0x0																											
								FFFFFE																							

MiniM encoding

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x9				0	0										
						11111					1111				0

3.2.29. OR De.r,Dx.r,De.r

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x3				S	0	I	DU																								
I=0								Rd					Rs1					Rs2					0	0	0	00000					0

MiniM core encoding

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x3				I	DU										
I=0						Rd			Rs1			Rs2			0

MiniM extended encoding

13	12	11	10	9	8	7	6	5	4	3	2	1	0
S	0												
		Rs1		Rs2		0	0	0	0	0	0	0	0

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x3				I	DU										
I=0						Rd			Rs1			Rs2			0

3.2.30. OR De.r,Dx.r,#I16

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x3				S	0	I	DU																								
I=1								Rd					16 Bit Immediate																M	SE	H

MiniM core encoding

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x3				I	DU										
I=1, SE5, H=0						Rd			5 Bit Immediate					M	SE

MiniM extended encoding

13	12	11	10	9	8	7	6	5	4	3	2	1	0
S	0												
		X16											H

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x3				I	DU										
I=1						Rd			X16					M	SE

3.2.31. RTH

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0xA				0x3																											
								011111111111111111111111																							

MiniM encoding

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x9				0xC				11							
										10		0xF			

3.2.32. RTI

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0xA				0x3																											
								111111111111111111111111																							

MiniM encoding

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x9				0xC				11							
										11		0xF			

3.2.33. SETD [A0.r+#S3],BBx.r

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0xB				0	1	I	0																								
			I=1					Rs					Rb					6-Bit Signed Imm						UA	BU		Ud				PP

MiniM core encoding

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0xA				00											
						Rs			Rb		S3			BUS	

I=1,S3 BU,Us=A0,BB UA,PP=0,0

MiniM extended encoding

13	12	11	10	9	8	7	6	5	4	3	2	1	0
	I												
1		0	0	Rb	S6			UA	BUA		Us		PP

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0xA				0 0											
		I=1				Rs			Rb		S6			Us	

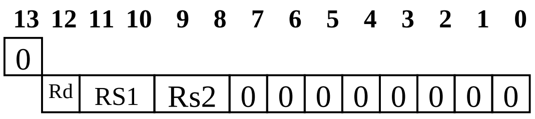
3.2.34. SUB Ae.r,Ax.r,Ae.r

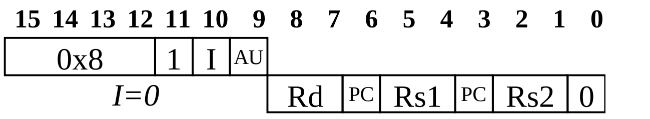
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x8				1	0	I	AU																								
			I=0					0	Rd				PC	Rs1				PC	Rs2				000000						0	0	0

MiniM core encoding

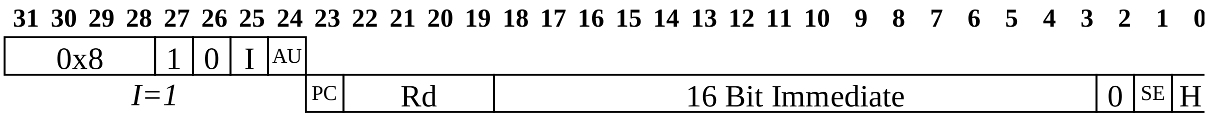
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x8				0	I	AU									
			I=0				Rd		PC	Rs1		PC	Rs2		0

MiniM extended encoding

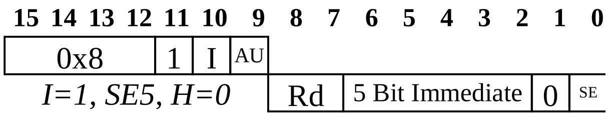




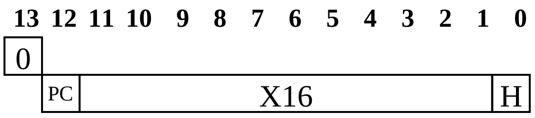
3.2.35. SUB Ae.e,Ax.r,#I16

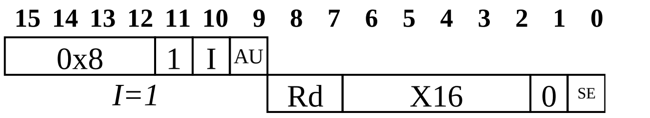


MiniM core encoding

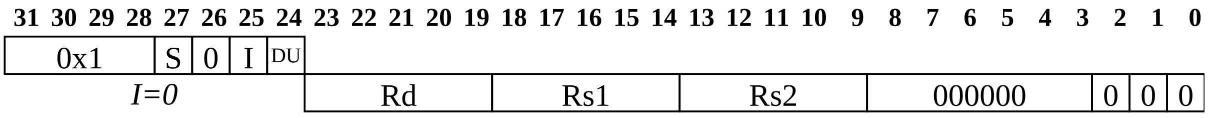


MiniM extended encoding

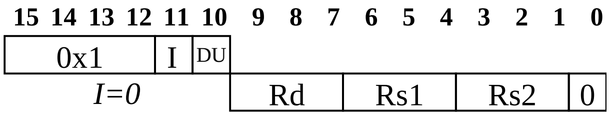




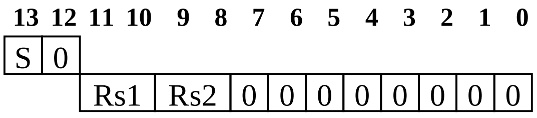
3.2.36. SUB De.r,Dx.r,De.r

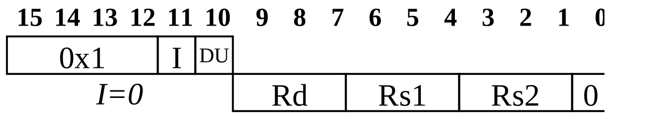


MiniM core encoding

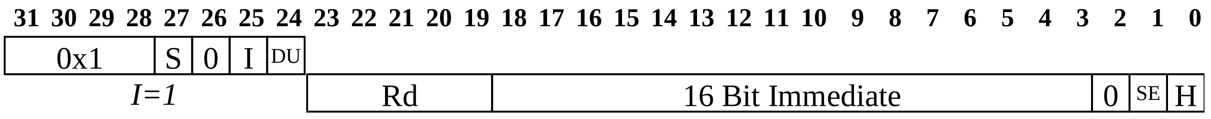


MiniM extended encoding

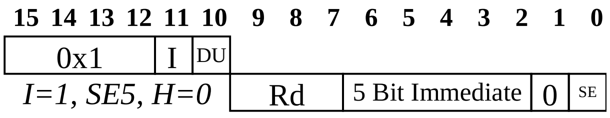




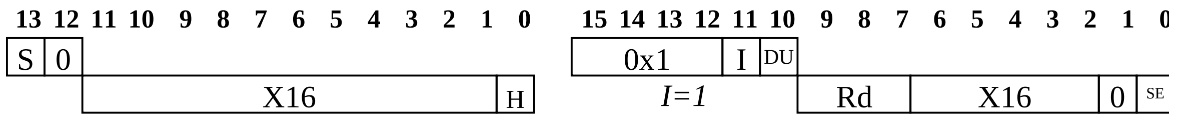
3.2.37. SUB De.e,Dx.r,#I16



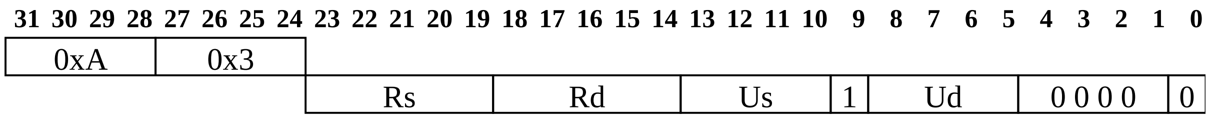
MiniM core encoding



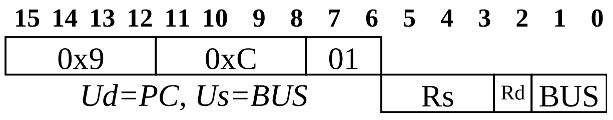
MiniM extended encoding



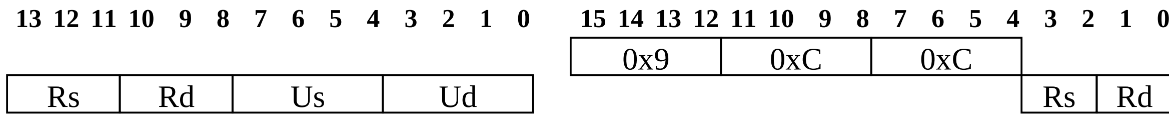
3.2.38. SWAP Ux.r,Uy.r



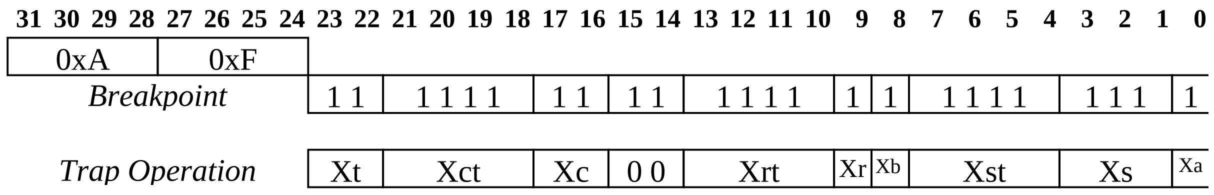
MiniM encoding



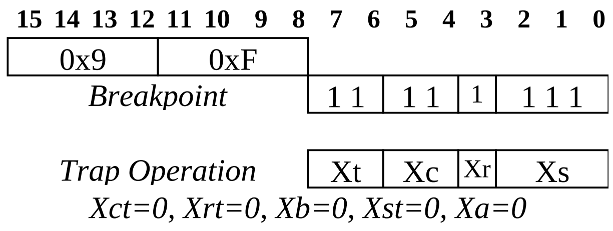
MiniM extended encoding



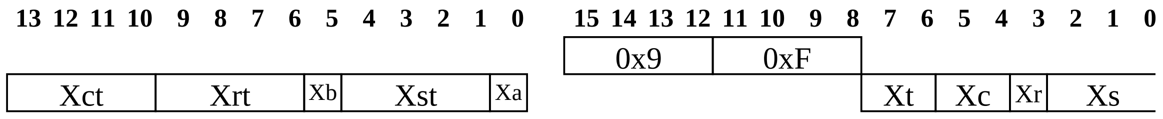
3.2.39. SWITCH #-1|0xC3020E|0xFF3FFF



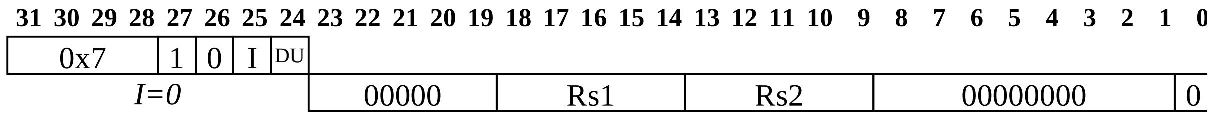
MiniM core encoding



MiniM extended encoding



3.2.40. TST Dx.r,De.r



MiniM core encoding

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x7				1	I	DU									
I=0							Rs1			Rs2					0

MiniM extended encoding

13	12	11	10	9	8	7	6	5	4	3	2	1	0
0													
	0	0	0	0	Rs1		0	0	0	0	0	0	0

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x7				1	I	DU									
I=0							Rs1			Rs2					0

3.2.41. TST Dx.r, #I16

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x7				1	0	I	DU																								
I=1								Rs					16 Bit Immediate																M	SE	H

MiniM core encoding

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x7				1	I	DU									
I=1,SE5,M=0,H=0							Rs			5 Bit					SE

MiniM extended encoding

13	12	11	10	9	8	7	6	5	4	3	2	1	0
0													
	X16											M	H

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x7				1	I	DU									
I=1							Rs1			X16					SE

3.2.42. TST Dx.r,Rx.r

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x7				1	0	0	DU																								
								00000					Rs1					Rs2					00000000								1

MiniM core encoding

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x7				1	0	DU									
							Rs1			Rs2					1

MiniM extended encoding

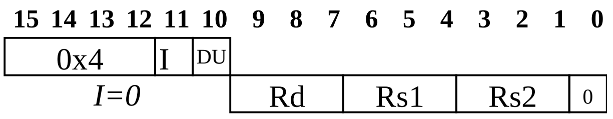
13	12	11	10	9	8	7	6	5	4	3	2	1	0
0													
	0	0	0	0	Rs1		0	0	0	0	0	0	1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x7				1	0	DU									
							Rs1			Rs2					1

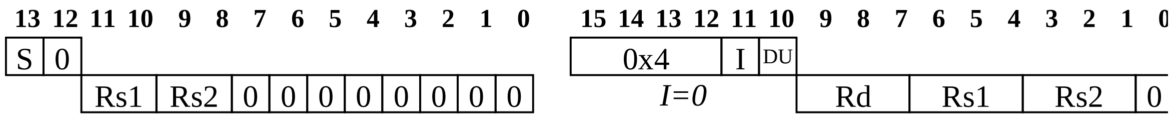
3.2.43. XOR De.r,Dx.r,De.r

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x4				S	0	I	DU																								
I=0								Rd					Rs1					Rs2					0	0	0	00000					0

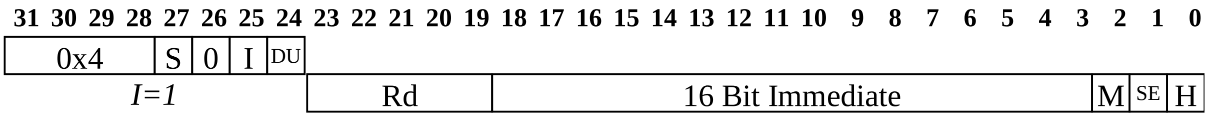
MiniM core encoding



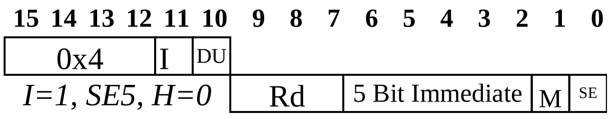
MiniM extended encoding



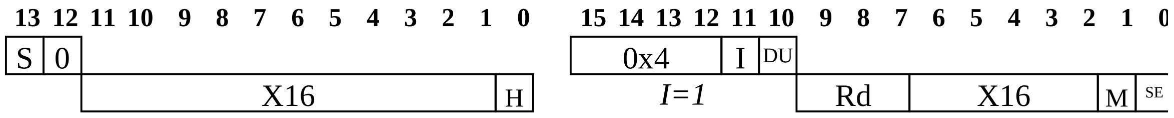
3.2.44. XOR De.e,Dx.r,#I16



MiniM core encoding

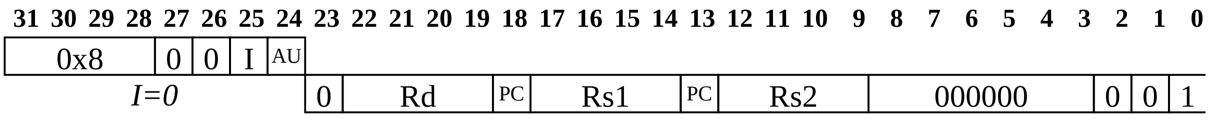


MiniM extended encoding



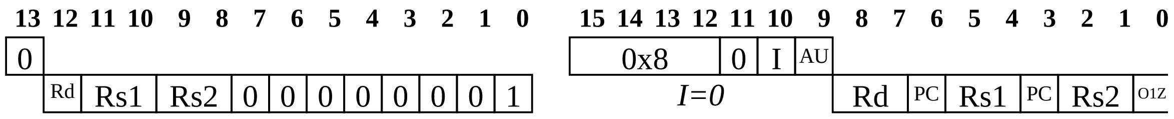
3.3. Extended general purpose instructions

3.3.1. ADD Ae.r,Ax.r,Rx.r

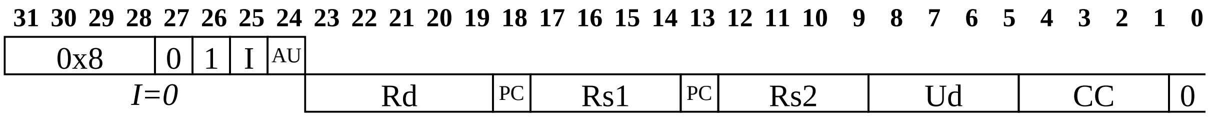


Note: PC+Rs2 is really a single 5-bit O2R Rs2 field

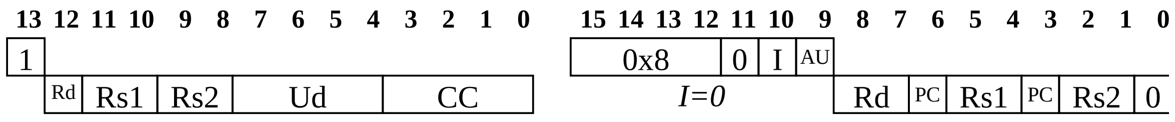
MiniM extended encoding



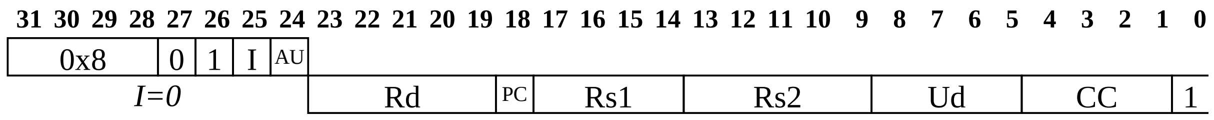
3.3.2. ADDcc Rx.r,Ax.r,Ae.r



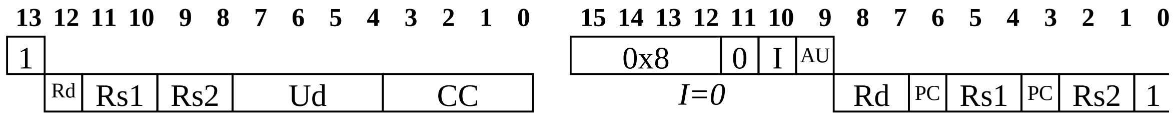
MiniM extended encoding



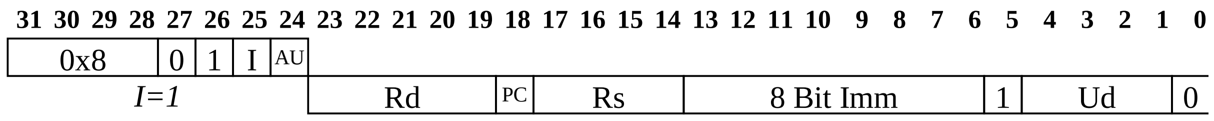
3.3.3. ADDcc Rx.r,Ax.r,Rx.r



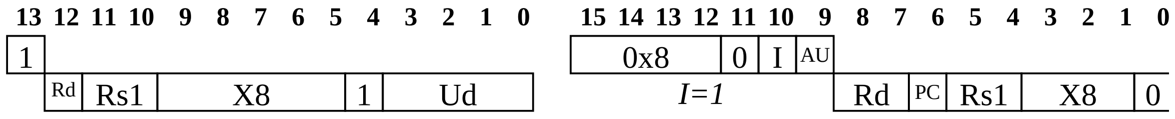
MiniM extended encoding



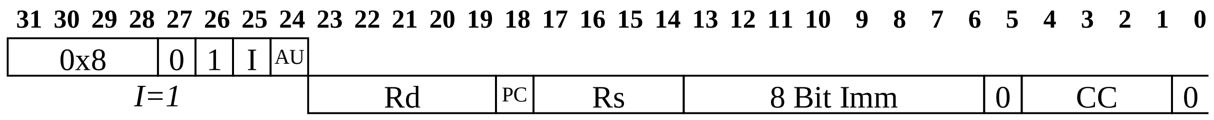
3.3.4. ADD Rx.r,Ax.r,#X8



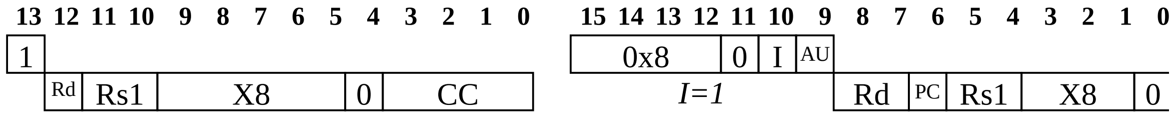
MiniM extended encoding



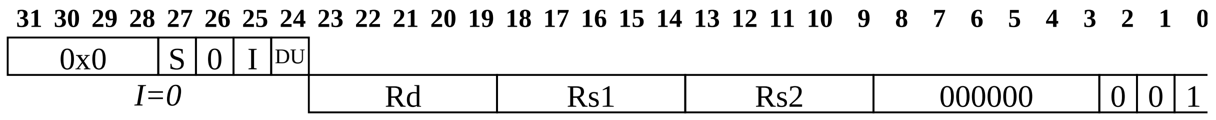
3.3.5. ADDcc Ae.r,Ax.r,#X8



MiniM extended encoding



3.3.6. ADD De.r,Dx.r,Rx.r



MiniM extended encoding

13	12	11	10	9	8	7	6	5	4	3	2	1	0
S	0												
		Rs1		Rs2		0	0	0	0	0	0	0	1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x0				I	DU										
		I=0				Rd			Rs1			Rs2			0

3.3.7. ADDcc Rx.r,Dx.r,De.r

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x0				S	1	I	DU																								
			I=0					Rd					Rs1					Rs2					Ud				CC				0

MiniM extended encoding

13	12	11	10	9	8	7	6	5	4	3	2	1	0
S	1												
		Rd		Rs2		Ud				CC			

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x0				I	DU										
		C = 1, I=0				Rd			Rs1			Rs2			0

3.3.8. ADDcc Rx.r,Dx.r,Rx.r

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x0				S	1	I	DU																								
			I=0					Rd					Rs1					Rs2					Ud				CC				1

MiniM extended encoding

13	12	11	10	9	8	7	6	5	4	3	2	1	0
S	1												
		Rd		Rs2		Ud				CC			

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x0				I	DU										
		C = 1, I=0				Rd			Rs1			Rs2			1

3.3.9. ADD Rx.r,Dx.r,#X8

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x0				S	1	I	DU																								
			I=1					Rd					Rs					8 Bit Imm								1	Ud				0

MiniM extended encoding

13	12	11	10	9	8	7	6	5	4	3	2	1	0
S	1												
		Rd		X8					1	Ud			

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x0				I	DU										
		I=1				Rd			Rs1			X8			0

3.3.10. ADDcc De.r,Dx.r,#X8

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x0				S	1	I	DU																								
			I=1					Rd					Rs					8 Bit Imm								0	CC				0

MiniM extended encoding

13	12	11–10	9–5	4	3–0
S	1				
		Rd	X8	0	CC

15–12	11	10	9–7	6–4	3–1	0
0x0	I	DU				
I=1			Rd	Rs1	X8	0

3.3.11. AND De.r,Dx.r,Rx.r

31–28	27	26	25	24	23–19	18–14	13–9	8	7	6	5–1	0
0x2	S	0	I	DU								
I=0					Rd	Rs1	Rs2	0	0	0	00000	1

MiniM extended encoding

13	12	11–10	9–8	7	6	5	4	3	2	1	0
S	0	Rs1	Rs2	0	0	0	0	0	0	0	1

15–12	11	10	9–7	6–4	3–1	0
0x2	I					
I=0		DU	Rd	Rs1	Rs2	0

3.3.12. ANDcc Rx.r,Dx.r,De.r

31–28	27	26	25	24	23–19	18–14	13–9	8–5	4–1	0
0x2	S	1	I	DU						
I=0					Rd	Rs1	Rs2	Ud	CC	0

MiniM extended encoding

13	12	11–10	9–8	7–4	3–0
S	1	Rd	Rs2	Ud	CC

15–12	11	10	9–7	6–4	3–1	0
0x2	I					
I=0		DU	Rd	Rs1	Rs2	0

3.3.13. ANDcc Rx.r,Dx.r,Rx.r

31–28	27	26	25	24	23–19	18–14	13–9	8–5	4–1	0
0x2	S	1	I	DU						
I=0					Rd	Rs1	Rs2	Ud	CC	1

MiniM extended encoding

13	12	11–8	7–5	4–0
1	S	Rd	Rs1	Rs2

15–12	11	10	9	8–5	4–1	0
0x2	I					
I=0		1	DU	Ud	CC	1

3.3.14. AND Rx.r,Dx.r,#X8

31–28	27	26	25	24	23–19	18–14	13–6	5	4–1	0
0x2	S	1	I	DU						
I=1					Rd	Rs	8 Bit Imm	1	Ud	1

MiniM extended encoding

13	12	11–10	9–5	4	3–0
S	1	Rd	X8	1	Ud

15–12	11	10	9–7	6–4	3–1	0
0x2	I	DU	Rd	Rs1	X8	0

I=1

3.3.15. ANDcc De.r,Dx.r,#X8

31–28	27	26	25	24	23–19	18–14	13–6	5	4–1	0
0x2	S	1	I	DU	Rd	Rs	8 Bit Imm	0	CC	0

I=1

MiniM extended encoding

13	12	11–10	9–5	4	3–0
S	1	Rd	X8	0	CC

15–12	11	10	9–7	6–4	3–1	0
0x2	I	DU	Rd	Rs1	X8	0

I=1

3.3.16. CMPcc Dx.r,De.r

31–28	27	26	25	24	23–19	18–14	13–9	8–5	4–1	0
0x7	0	1	I	DU	00000	Rs1	Rs2	0000	CC	0

I=0

MiniM extended encoding

13	12	11	10	9	8	7	6	5	4	3–0
1	0	0	0	0	0	0	0	0	0	CC

15–12	11	10	9	8–6	5–1	0
0x7	0	I	DU	Rs1	Rs2	0

I=0

3.3.17. CMPcc Dx.r,Rx.r

31–28	27	26	25	24	23–19	18–14	13–9	8–5	4–1	0
0x7	0	1	I	DU	00000	Rs1	Rs2	0000	CC	1

I=0

MiniM extended encoding

13	12	11	10	9	8	7	6	5	4	3–0
1	0	0	0	0	0	0	0	0	0	CC

15–12	11	10	9	8–6	5–1	0
0x7	0	I	DU	Rs1	Rs2	1

I=0

3.3.18. CMPcc Dx.r,#X8

31–28	27	26	25	24	23–19	18–14	13–6	5	4–1	0
0x7	0	1	I	DU	00000	Rs	8 Bit Imm	0	CC	0

I=1

MiniM extended encoding

13	12	11	10	9–5	4	3–0
1	0	0	0	X8	0	CC

15–12	11	10	9	8	7	6–4	3–1	0
0x7	0	I	DU	0	0	Rs1	X8	0

I=1

3.3.19. CPRx [BBxUaRoPp|BBxUaS6Pp],CP.n

31-28	27	26	25	24	23-19	18-14	13-9	8	7	6-5	4-1	0
0xB	0	L2	I	L1								
I=0					Rs	Rb	Ro	0	UA	BU	0110	PP
I=1					Rs	Rb	6-Bit Signed Imm		UA	BU	0110	PP

MiniM Extended Encoding

13	12	11	10	9	8	7	6-4	3	2-1	0
	I									
L2		L1	0	Rb	0	0	Ro	UA	Us	PP
L2		L1	0	Rb	S6					

13	12	11	10	9	8-6	5	4-3	2-1	0
L2		L1	0	Rb	S6	UA	BUA	Us	PP

15-12	11-10	9-7	6-5	4	3-2	1-0
0xA	0 0					
I=0		Rs	Rb	0	BUA	Us
I=1		Rs	Rb	S6		Us

3.3.20. GETL BBx.r,BBo.e,[BBxUaRoPp|BBxUaS6Pp]

31-28	27	26	25	24	23-19	18-14	13-9	8	7	6-5	4-1	0
0xC	0	1	I	1								
I=0					Rd	Rb	Ro	0	UA	BU	Ud	PP
I=1					Rd	Rb	6-Bit Signed Imm		UA	BU	Ud	PP

MiniM extended encoding

13	12	11	10	9	8-6	5	4	3	2-1	0
	I									
1		1	Rd	Rb	Ro			UA	Ud	PP
1		1	Rd	Rb	S6	ua	BUA		Ud	PP

15-12	11-10	9-7	6-5	4	3-2	1-0
0xA	1 0					
I=0		Rd	Rb	0	BUA	Ud
I=1		Rd	Rb	S6		Ud

3.3.21. GETx Ux.r,[BBxUaRoPp|BBxUaS6Pp]

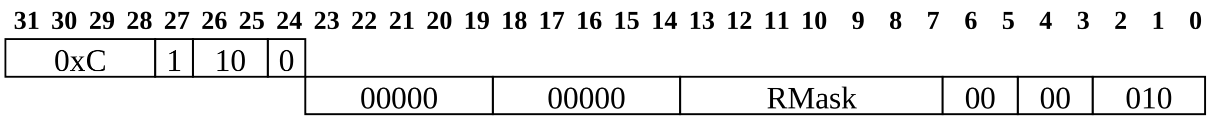
31-28	27	26	25	24	23-19	18-14	13-9	8	7	6-5	4-1	0
0xC	0	L2	I	L1								
I=0					Rd	Rb	Ro	0	UA	BU	Ud	PP
I=1					Rd	Rb	6-Bit Signed Imm		UA	BU	Ud	PP

MiniM extended encoding

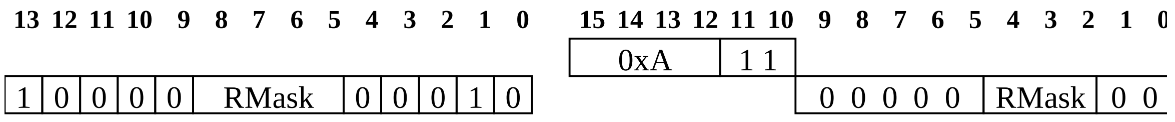
13	12	11	10	9	8-6	5	4	3	2-1	0
	I									
L2		L1	Rd	Rb	Ro			UA	Ud	PP
L2		L1	Rd	Rb	S6	ua	BUA		Ud	PP

15-12	11-10	9-7	6-5	4	3-2	1-0
0xA	1 0					
I=0		Rd	Rb	0	BUA	Ud
I=1		Rd	Rb	S6		Ud

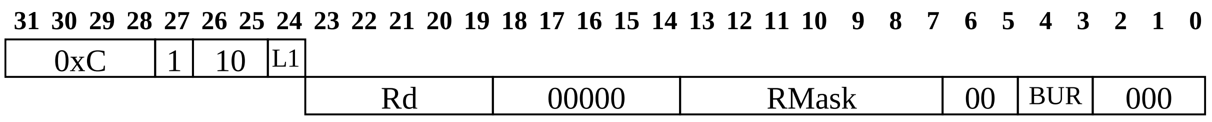
3.3.22. MDRD RD..RD..



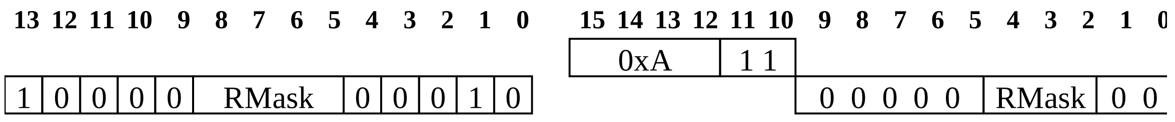
MiniM extended encoding



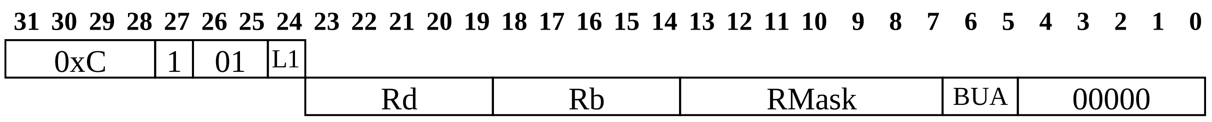
3.3.23. MMOVx BBx.r..BBe.e..,RD



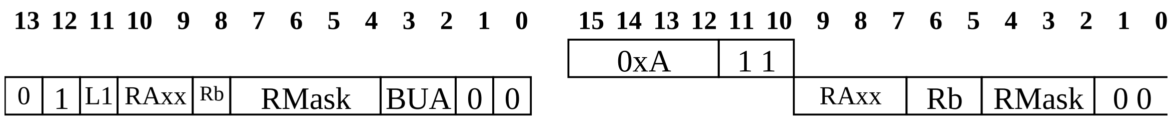
MiniM extended encoding



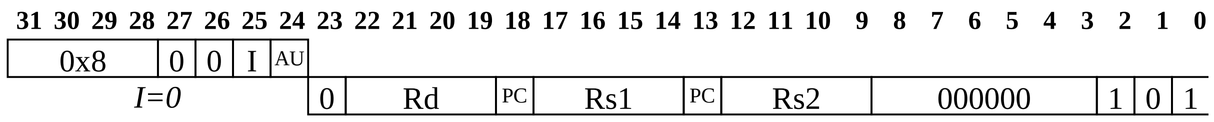
3.3.24. MMOVx RAxx..RAee..,[BBx.r]



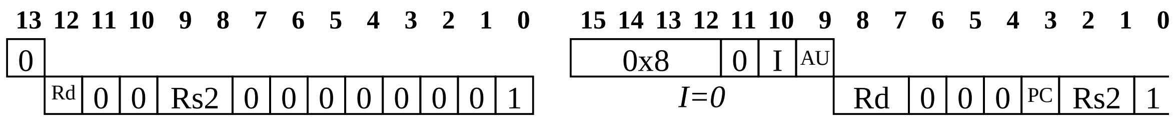
MiniM extended encoding



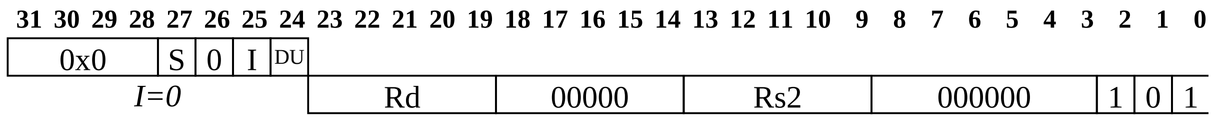
3.3.25. MOV Ae.r,Rx.r



MiniM extended encoding



3.3.26. MOV Dx.r,Rx.r



MiniM extended encoding

	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	S	0												
			0	0	Rs2		0	0	0	0	0	0	0	1

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	0x0				I	DU										
I=0							Rd			000			Rs2			1

3.3.27. MOVx RAxx,[BBxUaRoPp|BBxUaS6Pp]

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	0xC				0	L2	I	L1																								
I=0									RAxx					Rb					Ro					0	UA	BU		0110				PP
I=1									RAxx					Rb					6-Bit Signed Imm						UA	BU		0110				PP

MiniM extended encoding

	13	12	11	10	9	8	7	6	5	4	3	2	1	0
		I												
	L2		L1	Rd	Rb	Ro					UA	0 1		PP
	L2		L1	Rd	Rb	S6			ua	BUA		0 1		PP

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	0xA				1 0											
I=0							Rd			Rb		0	BUA		1 0	
I=1							Rd			Rb		S6			1 0	

3.3.28. MUL De.r,Dx.r,Rx.r

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	0x6				0	0	I	DU																								
I=0									Rd					Rs1					Rs2					00		L2	00000					1

MiniM extended encoding

	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	0	0												
			Rs1		Rs2		0	0	0	0	0	0	0	1

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	0x6				I	DU										
I=0							Rd			Rs1			Rs2			L2

3.3.29. MUL BBx.r,Dx.r,De.r

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	0x6				0	1	I	DU																								
I=0									Rd					Rs1					Rs2					00		L2	1	Ud				0

MiniM extended encoding

	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	0	1												
			Rd		Rs2		0	0	L2	1	Ud			

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	0x6				I	DU										
I=0							Rd			Rs1			Rs2			0

3.3.30. MUL BBx.r,Dx.r,Rx.r

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	0x6				0	1	I	DU																								
I=0									Rd					Rs1					Rs2					00		L2	1	Ud				1

MiniM extended encoding

13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1												
		Rd		Rs2		0	0	L2	1	Ud			

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x6				I	DU										
I=0						Rd			Rs1			Rs2			1

3.3.31. MUL BBx.r,Dx.r,#X8

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x6				0	1	I	DU																								
I=1								Rd					Rs					8 Bit Imm								1	Ud				L2

MiniM extended encoding

13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1												
		Rd		X8					1	Ud			

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x6				I	DU										
I=1						Rd			Rs1			X8			L2

3.3.32. MULcc De.r,Dx.r,De.r

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x6				0	1	I	DU																								
I=0								Rd					Rs1					Rs2					00		L2	0	CC				0

MiniM extended encoding

13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1												
		Rd		Rs2		0	0	L2	0	CC			

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x6				I	DU										
I=0						Rd			Rs1			Rs2			0

3.3.33. MULcc De.r,Dx.r,Rx.r

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x6				0	1	I	DU																								
I=0								Rd					Rs1					Rs2					00		L2	0	CC				1

MiniM extended encoding

13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1												
		Rd		Rs2		0	0	L2	0	CC			

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x6				I	DU										
I=0						Rd			Rs1			Rs2			1

3.3.34. MULcc De.r,Dx.r,#X8

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x6				0	1	I	DU																								
I=1								Rd					Rs					8 Bit Imm								0	CC				L2

MiniM extended encoding

13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1												
		Rd		X8					0	CC			

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x6				I	DU										
			I=1			Rd			Rs1			X8			L2

3.3.35. NEG Ae.r,Rx.r

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x8				1	0	I	AU																								
				I=0				0	Rd				PC	Rs1				PC	Rs2				000000						1	0	1

MiniM extended encoding

13	12	11	10	9	8	7	6	5	4	3	2	1	0
0													
	Rd	0	0	Rs2		0	0	0	0	0	0	0	1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x8				1	I	AU									
			I=0				Rd		0	0	0	PC	Rs2		1

3.3.36. NEG De.r,Rx.r

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x1				S	0	I	DU																								
				I=0				Rd					00000					Rs2					000000						1	0	1

MiniM extended encoding

13	12	11	10	9	8	7	6	5	4	3	2	1	0
S	0												
		0	0	Rs2		0	0	0	0	0	0	0	1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x1				I	DU										
			I=0			Rd			000			Rs2			1

3.3.37. OR De.r,Dx.r,Rx.r

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x3				S	0	I	DU																								
				I=0				Rd					Rs1					Rs2					0	0	0	00000					1

MiniM extended encoding

13	12	11	10	9	8	7	6	5	4	3	2	1	0
S	0	Rs1		Rs2		0	0	0	0	0	0	0	1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x3				I											
	I=0				DU	Rd			Rs1			Rs2			0

3.3.38. ORcc Rx.r,Dx.r,De.r

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x3				S	1	I	DU																								
				I=0				Rd					Rs1					Rs2					Ud				CC				0

MiniM extended encoding

13	12	11	10	9	8	7	6	5	4	3	2	1	0
S	1	Rd		Rs2		Ud				CC			

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x3				I											
		I=0			DU	Rd			Rs1			Rs2			0

3.3.39. ORcc Rx.r,Dx.r,Rx.r

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x3				S	1	I	DU																								
			I=0					Rd					Rs1					Rs2					Ud				CC				1

MiniM extended encoding

13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	S	Rd				Rs1			Rs2				

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x3				I											
		I=0			1	DU	Ud				CC				1

3.3.40. OR Rx.r,Dx.r,#X8

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x3				S	1	I	DU																								
			I=1					Rd					Rs					8 Bit Imm								1	Ud				1

MiniM extended encoding

13	12	11	10	9	8	7	6	5	4	3	2	1	0
S	1	Rd		X8					1	Ud			

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x3				I											
		I=1			DU	Rd			Rs1			X8			0

3.3.41. ORcc De.r,Dx.r,#X8

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x3				S	1	I	DU																								
			I=1					Rd					Rs					8 Bit Imm								0	CC				0

MiniM extended encoding

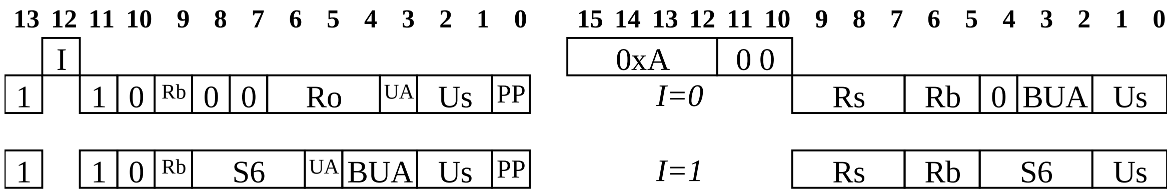
13	12	11	10	9	8	7	6	5	4	3	2	1	0
S	1	Rd		X8					0	CC			

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x3				I											
		I=1			DU	Rd			Rs1			X8			0

3.3.42. SETL [BBxUaRoPp|BBxUaS6Pp],BBx.r,BBo.e

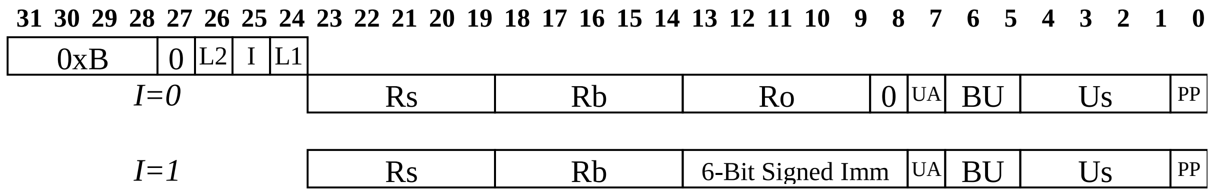
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0xB				0	1	I	1																								
			I=0					Rs					Rb					Ro					0	UA	BU		Us				PP
			I=1					Rs					Rb					6-Bit Signed Imm						UA	BU		Us				PP

MiniM extended encoding

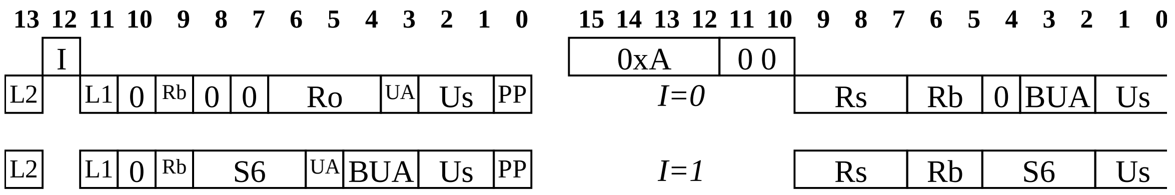


13 12 11 10 9 8 7 6 5 4 3 2 1 0
I
1 | 1 | 0 | Rb | 0 | 0 | Ro | UA | Us | PP
1 | 1 | 0 | Rb | S6 | UA | BUA | Us | PP
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0xA | 0 0
I=0 | Rs | Rb | 0 | BUA | Us
I=1 | Rs | Rb | S6 | Us

3.3.43. SETx [BBxUaRoPp|BBxUaS6Pp],Ux.r

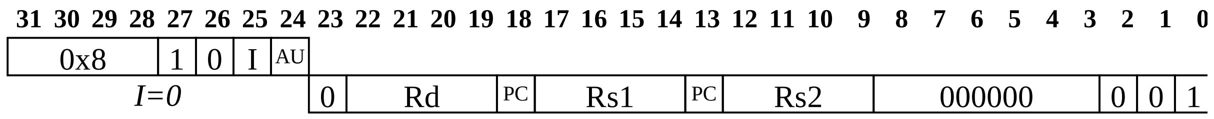


MiniM extended encoding



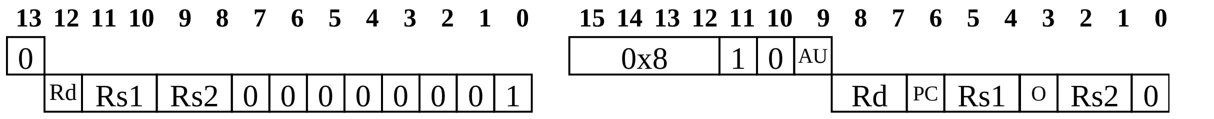
13 12 11 10 9 8 7 6 5 4 3 2 1 0
I
L2 | L1 | 0 | Rb | 0 | 0 | Ro | UA | Us | PP
L2 | L1 | 0 | Rb | S6 | UA | BUA | Us | PP
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0xA | 0 0
I=0 | Rs | Rb | 0 | BUA | Us
I=1 | Rs | Rb | S6 | Us

3.3.44. SUB Ae.r,Ax.r,Rx.r

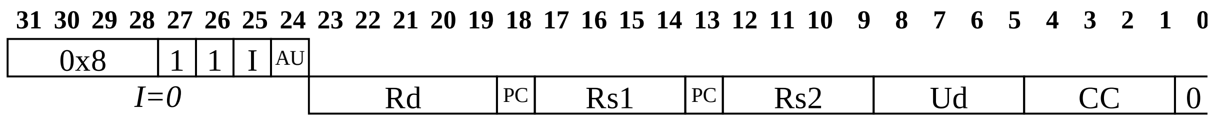


Note: PC+Rs2 is really a single 5-bit O2R Rs2 field

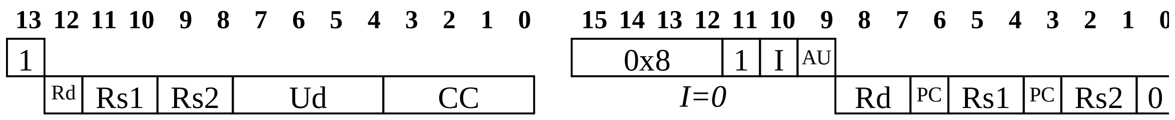
MiniM extended encoding



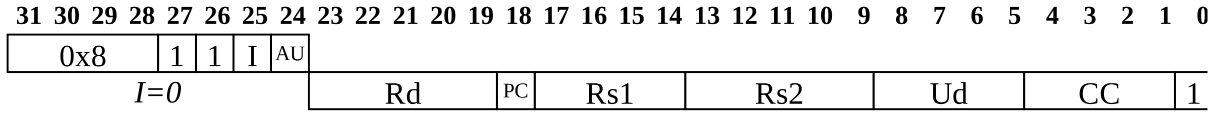
3.3.45. SUBcc Rx.r,Ax.r,Ae.r



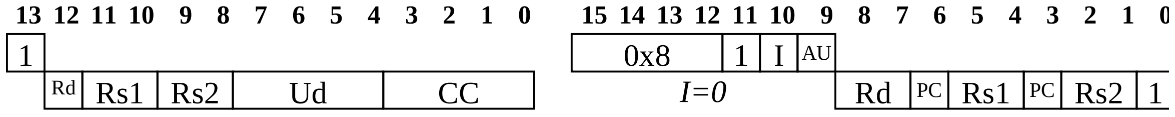
MiniM extended encoding



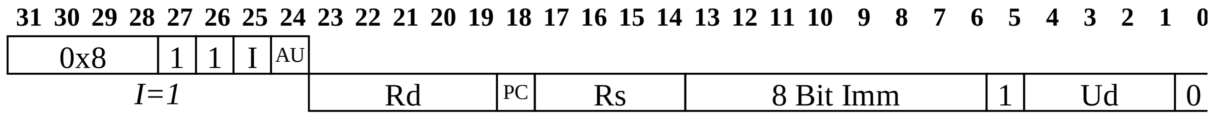
3.3.46. SUBcc Rx.r,Ax.r,Rx.r



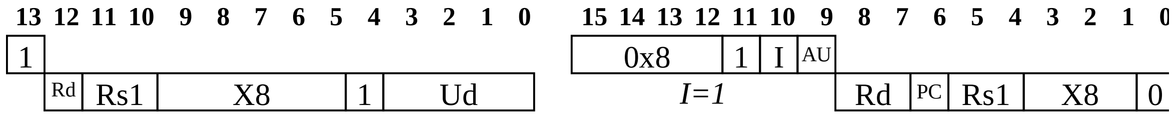
MiniM extended encoding



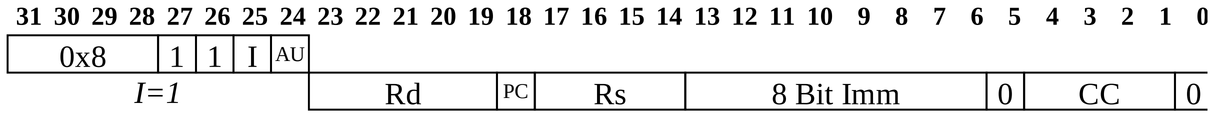
3.3.47. SUB Rx.r,Ax.r,#X8



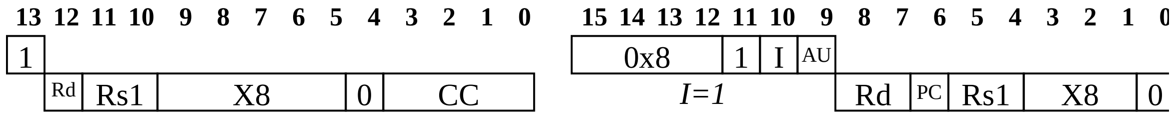
MiniM extended encoding



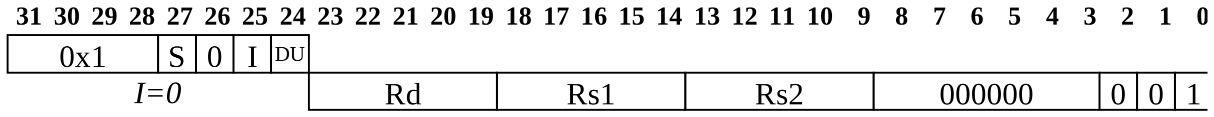
3.3.48. SUBcc Ae.r,Ax.r,#X8



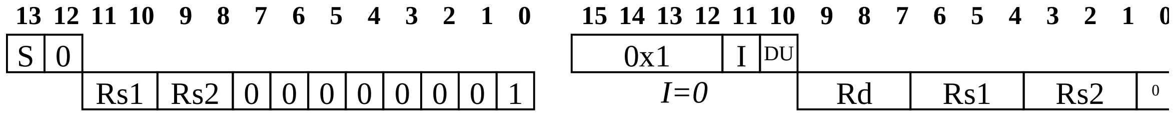
MiniM extended encoding



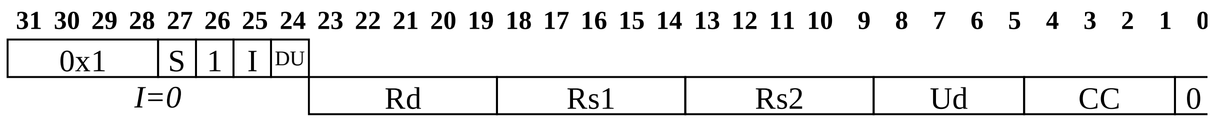
3.3.49. SUB De.r,Dx.r,Rx.r



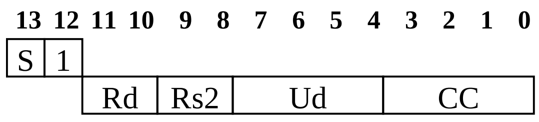
MiniM extended encoding

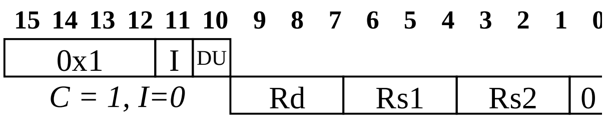


3.3.50. SUBcc Rx.r,Dx.r,De.r

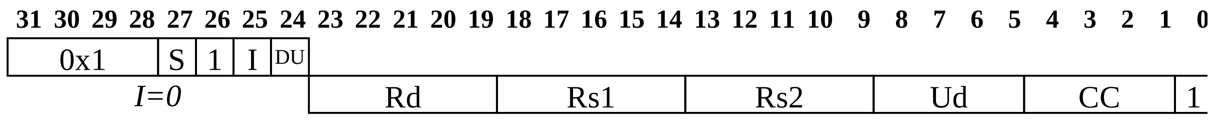


MiniM extended encoding

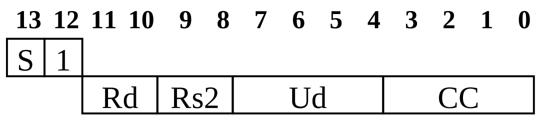


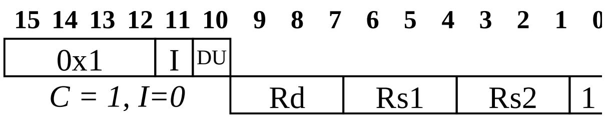


3.3.51. SUBcc Rx.r,Dx.r,Rx.r

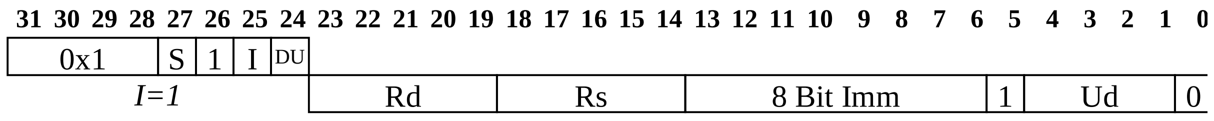


MiniM extended encoding

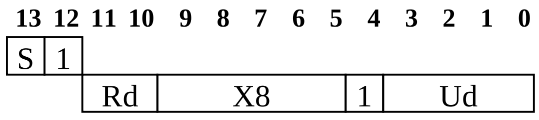


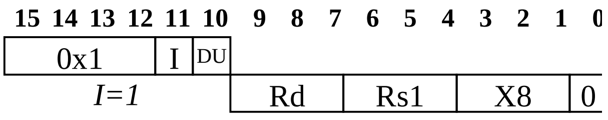


3.3.52. SUB Rx.r,Dx.r,#X8

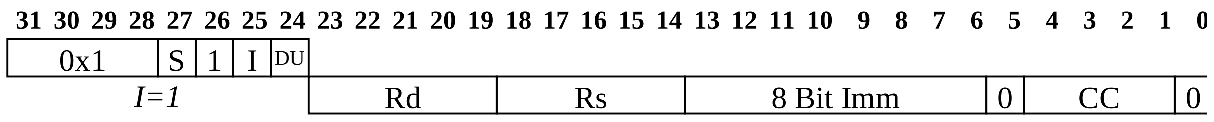


MiniM extended encoding

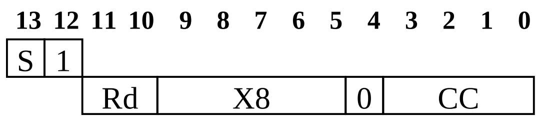


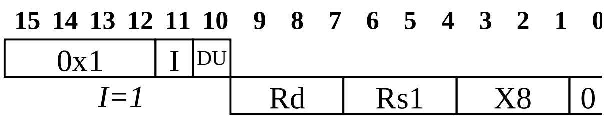


3.3.53. SUBcc De.r,Dx.r,#X8

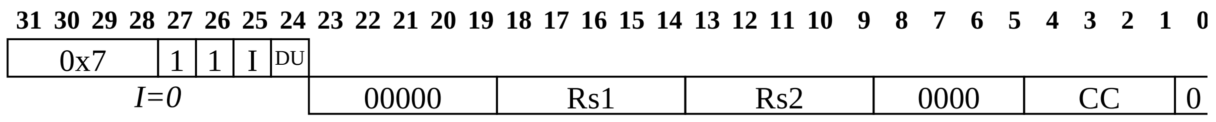


MiniM extended encoding

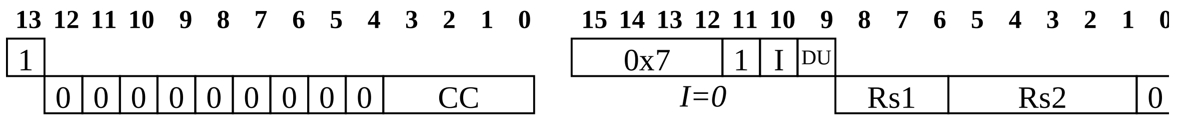




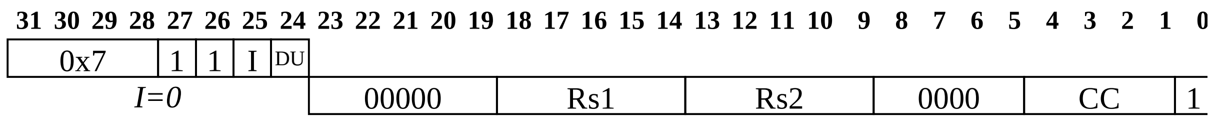
3.3.54. TSTcc Dx.r,De.r



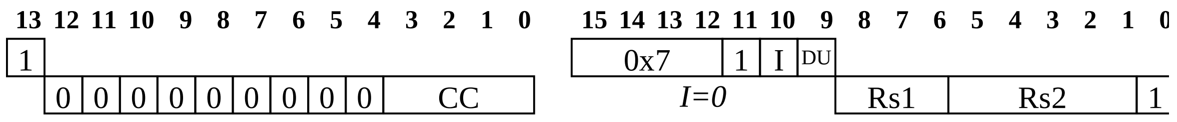
MiniM extended encoding



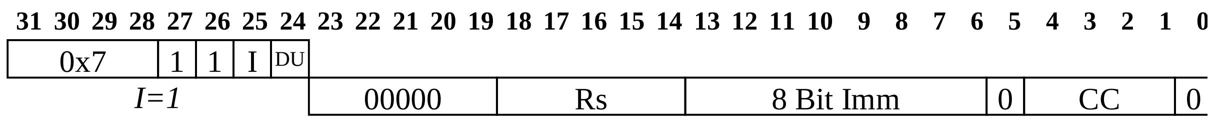
3.3.55. TSTcc Dx.r,Rx.r



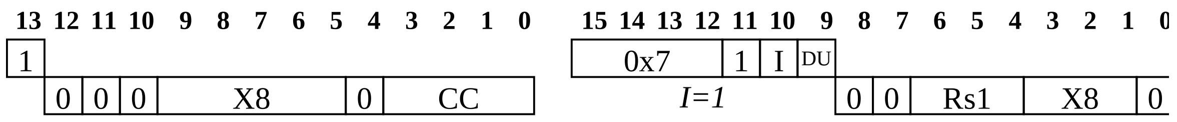
MiniM extended encoding



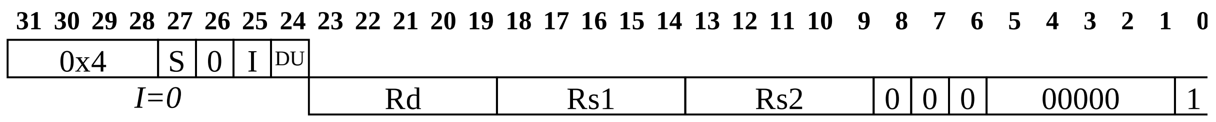
3.3.56. TSTcc Dx.r,#X8



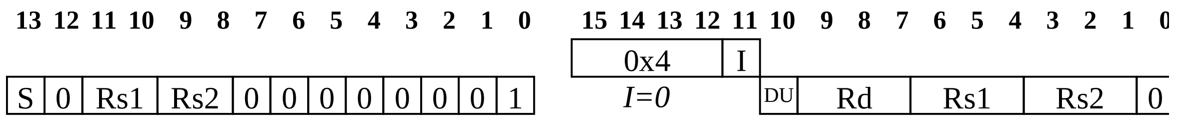
MiniM extended encoding



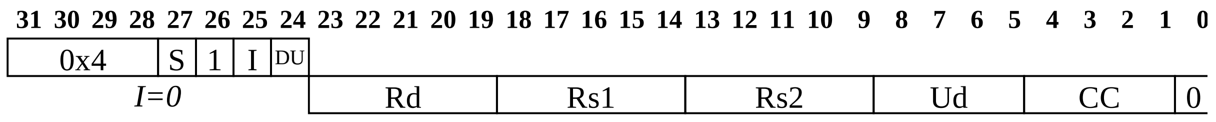
3.3.57. XOR De.r,Dx.r,Rx.r



MiniM extended encoding



3.3.58. XORcc Rx.r,Dx.r,De.r



MiniM extended encoding

13	12	11	10	9	8	7	6	5	4	3	2	1	0
S	1	Rd		Rs2		Ud				CC			

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x4				I											
	I=0				DU	Rd			Rs1			Rs2			0

3.3.59. XORcc Rx.r,Dx.r,Rx.r

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x4				S	1	I	DU																								
			I=0					Rd					Rs1					Rs2					Ud				CC				1

MiniM extended encoding

13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	S	Rd				Rs1			Rs2				

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x4				I											
	I=0				1	DU	Ud				CC				1

3.3.60. XOR Rx.r,Dx.r,#X8

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x4				S	1	I	DU																								
			I=1					Rd					Rs					8 Bit Imm								1	Ud				1

MiniM extended encoding

13	12	11	10	9	8	7	6	5	4	3	2	1	0
S	1	Rd		X8					1	Ud			

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x4				I											
	I=1				DU	Rd			Rs1			X8			0

3.3.61. XORcc De.r,Dx.r,#X8

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x4				S	1	I	DU																								
			I=1					Rd					Rs					8 Bit Imm								0	CC				0

MiniM extended encoding

13	12	11	10	9	8	7	6	5	4	3	2	1	0
S	1	Rd		X8					0	CC			

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x4				I											
	I=1				DU	Rd			Rs1			X8			0

3.4. Data unit extended instructions

3.4.1. ABS De.r,Dx.r

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x7				000			DU																								
								Rd					Rs					00000000								1	0	10		00	

MiniM extended encoding

13	12	11	10	9	8	7	6	5	4	3	2	1	0
0													
	Rd			Rs		0	0	0	1010				0

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x7				0	0	DU									
							Rs			00000					0

3.4.2. FFB De.r,Dx.r

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x7				000			DU																								
								Rd					Rs					00000000								0	0	01		00	

MiniM encoding

13	12	11	10	9	8	7	6	5	4	3	2	1	0
0													
	Rd			Rs		0	0	0	0001				0

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x7				0	0	DU									
							Rs			00000					0

3.4.3. MAX De.r,Dx.r,De.r

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x7				000			DU																								
								Rd					Rs1					Rs2					000			1	0	01		00	

MiniM encoding

13	12	11	10	9	8	7	6	5	4	3	2	1	0
0													
	Rd			Rs1		0	0	0	1001				0

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x7				0	0	DU									
							Rs1			Rs2					0

3.4.4. MIN De.r,Dx.r,De.r

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x7				000			DU																								
								Rd					Rs1					Rs2					000			1	0	00		00	

MiniM extended encoding

13	12	11	10	9	8	7	6	5	4	3	2	1	0
0													
	Rd			Rs1		0	0	0	1000				0

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x7				0	0	DU									
							Rs1			Rs2					0

3.4.5. MORT De.r,Dx.r,De.r

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x7				000			DU																								
								Rd					Rs1					Rs2					001			0	0	11		0	0

MiniM extended encoding

13	12	11	10	9	8	7	6	5	4	3	2	1	0
0													
	Rd			Rs1		0	1	0	0110				0

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x7				0	0	DU									
							Rs1			Rs2					0

3.4.6. MORT De.r,Dx.r,Rx.r

31 30 29 28	27 26 25	24	23 22 21 20 19	18 17 16 15 14	13 12 11 10 9	8 7 6	5	4	3 2	1	0
0x7	000	DU	Rd	Rs1	Rs2	001	0	0	11	0	1

MiniM extended encoding

13	12 11 10	9 8	7	6	5	4 3 2 1	0
0	Rd	Rs1	0	1	0	0110	0

15 14 13 12	11	10	9	8 7 6	5 4 3 2 1	0
0x7	0	0	DU	Rs1	Rs2	1

3.4.7. NMIN De.r,Dx.r,De.r

31 30 29 28	27 26 25	24	23 22 21 20 19	18 17 16 15 14	13 12 11 10 9	8 7 6	5	4	3 2	1 0
0x7	000	DU	Rd	Rs1	Rs2	000	1	0	11	00

MiniM extended encoding

13	12 11 10	9 8	7	6	5	4 3 2 1	0
0	Rd	Rs1	0	0	0	1011	0

15 14 13 12	11	10	9	8 7 6	5 4 3 2 1	0
0x7	0	0	DU	Rs1	Rs2	0

3.4.8. NORM De.r,Dx.r

31 30 29 28	27 26 25	24	23 22 21 20 19	18 17 16 15 14	13 12 11 10 9 8 7 6	5	4	3 2	1 0
0x7	000	DU	Rd	Rs	00000000	0	0	10	00

MiniM extended encoding

13	12 11 10	9 8	7	6	5	4 3 2 1	0
0	Rd	Rs	0	0	0	0010	0

15 14 13 12	11	10	9	8 7 6	5 4 3 2 1	0
0x7	0	0	DU	Rs	00000	0

3.4.9. VPACK De.r,Dx.r,De.r

31 30 29 28	27 26 25	24	23 22 21 20 19	18 17 16 15 14	13 12 11 10 9	8 7 6	5	4	3 2	1	0
0x7	000	DU	Rd	Rs1	Rs2	000	0	0	11	0	0

MiniM extended encoding

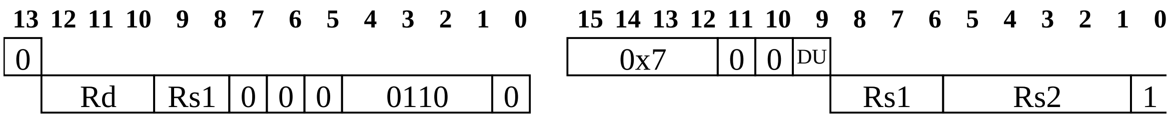
13	12 11 10	9 8	7	6	5	4 3 2 1	0
0	Rd	Rs1	0	0	0	0110	0

15 14 13 12	11	10	9	8 7 6	5 4 3 2 1	0
0x7	0	0	DU	Rs1	Rs2	0

3.4.10. VPACK De.r,Dx.r,Rx.r

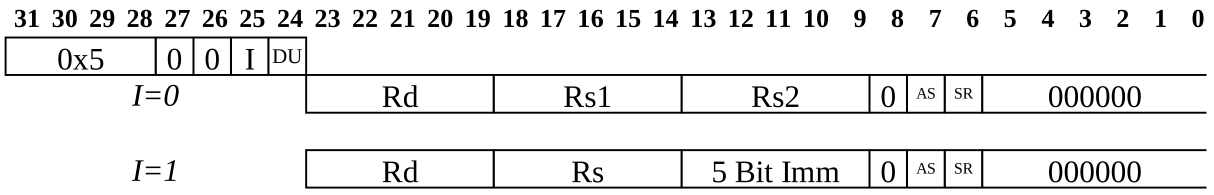
31 30 29 28	27 26 25	24	23 22 21 20 19	18 17 16 15 14	13 12 11 10 9	8 7 6	5	4	3 2	1	0
0x7	000	DU	Rd	Rs1	Rs2	000	0	0	11	0	1

MiniM extended encoding

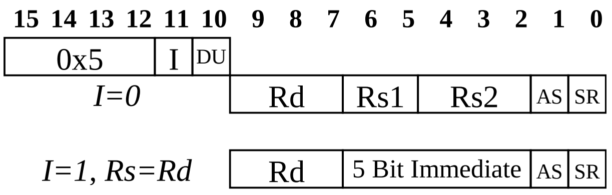


3.5. Shift instructions

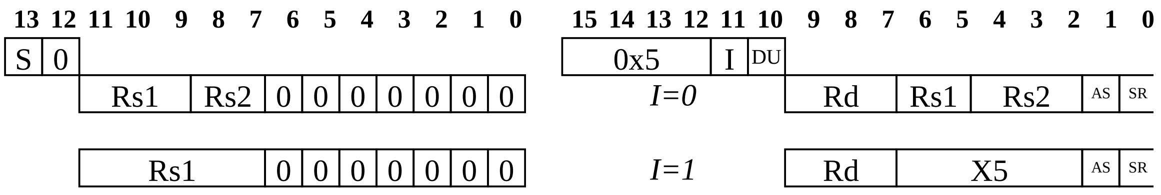
3.5.1. SHIFT De.r,Dx.r,De.r|#X5



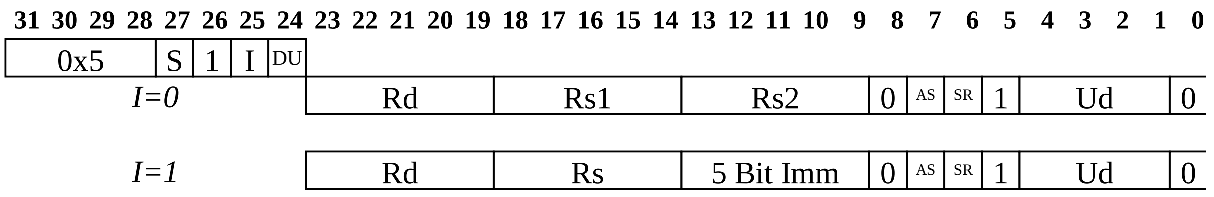
MiniM core encoding



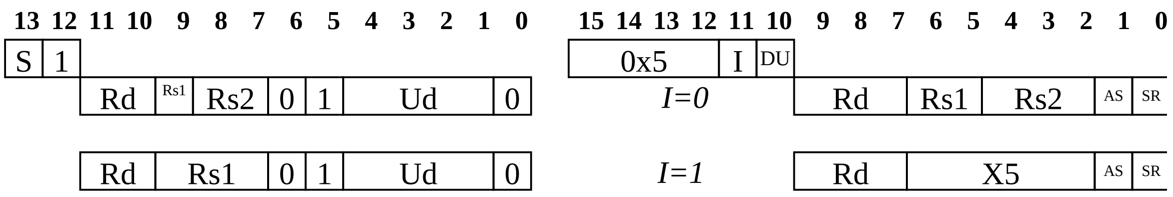
MiniM extended encoding



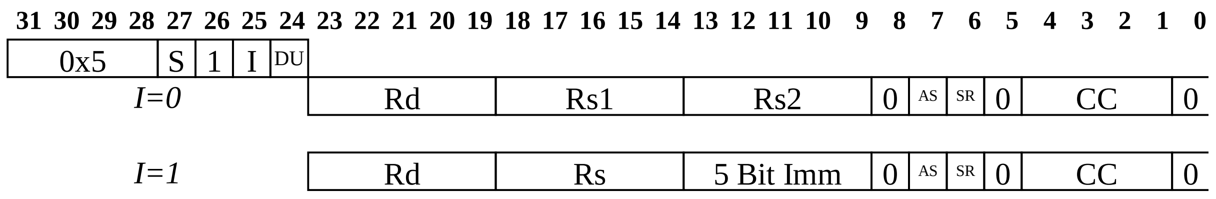
3.5.2. SHIFT Rx.r,Dx.r,De.r|#X5



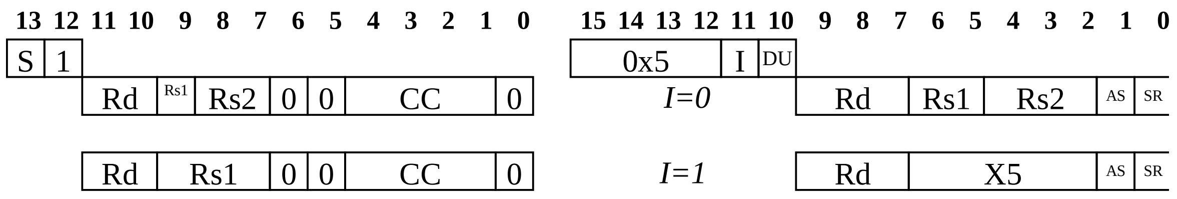
MiniM extended encoding



3.5.3. SHIFTcc De.r,Dx.r,De.r|#X5

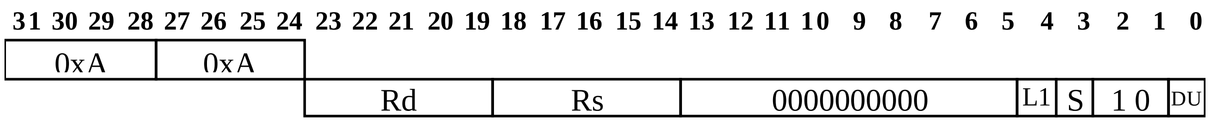


MiniM extended encoding

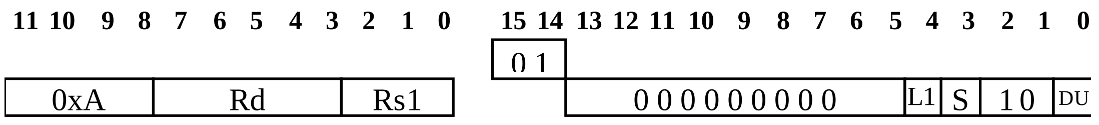


3.6. Long general purpose instructions

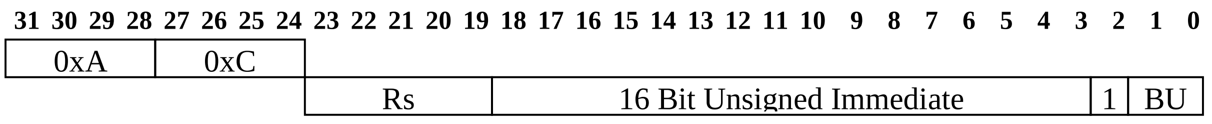
3.6.1. BEXx De.r,Dx.r



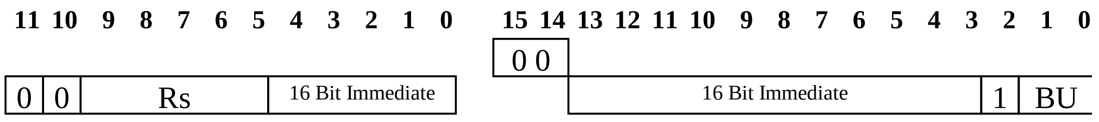
MiniM long encoding



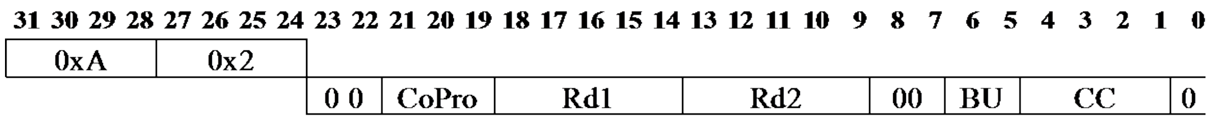
3.6.2. CALL BBx.r,#X16



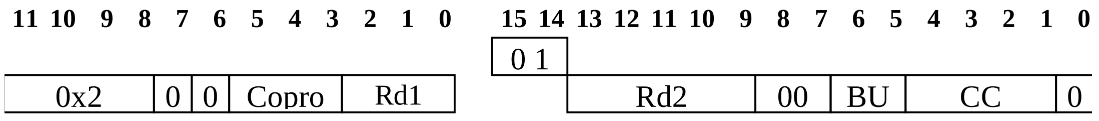
MiniM long encoding



3.6.3. CPRLcc BBx.r,BBo.r,CP.n



MiniM long encoding



3.6.4. CPRR [BBxUaRoPp],CP.r

31 30 29 28	27	26	25	24	23 22	21 20 19	18 17 16 15 14	13 12 11 10 9	8 7 6 5 4	3 2	1 0
0xD	0	UD	L1	PP	0 0	CoPro	0 0 0 0 0	Rdb	Rdo	BUD	BUD

MiniM long encoding

11	10	9	8	7	6	5 4 3	2 1 0
0	UD	L1	PP	0	0	Copro	0 0 0

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

3.6.5. CPRxcc Ux.r,CP.n

31 30 29 28	27 26 25 24	23 22	21 20 19	18 17 16 15 14	13 12 11 10	9	8 7 6 5	4 3 2 1	0
0xA	0x1	0 0	CoPro	Rd	0000	L2	Ud	CC	L1

MiniM long encoding

11 10 9 8	7	6 5 4	3 2 1 0
0x1	0	Copro	Rd

15 14	13 12 11 10	9	8 7 6 5	4 3 2 1	0
0 1	0000	L2	Ud	CC	L1

3.6.6. CPRxcc [BBx.r],CP.n

31 30 29 28	27 26 25 24	23 22	21 20 19	18 17 16 15 14	13 12 11 10	9	8 7	6 5	4 3 2 1	0
0xA	0x4	0 0	CoPro	Rb	Us	L2	00	BU	CC	L1

MiniM long encoding

11 10 9 8	7	6	5 4 3	2 1 0
0x4	0	0	CoPro	Rb

15 14	13 12 11 10	9	8 7	6 5	4 3 2 1	0
0 1	0 1 1 0	L2	00	BU	CC	L1

3.6.7. CPWL CP.n,BBx.r,BBo.r|#X16

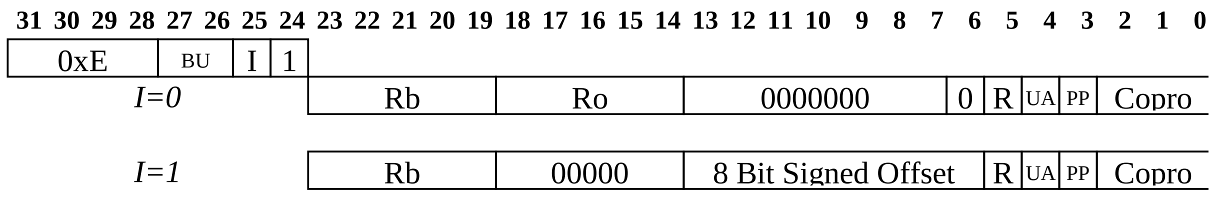
	31 30 29 28	27 26	25	24	23 22 21 20 19	18 17 16 15 14	13 12 11 10 9 8 7 6 5 4 3	2 1 0
	0xE	BU	I	0				
I=0					Rs1	Rs2	00000000000	Copro
I=1					Rs1	16 Bit Unsigned Immediate		Copro

MiniM long encoding

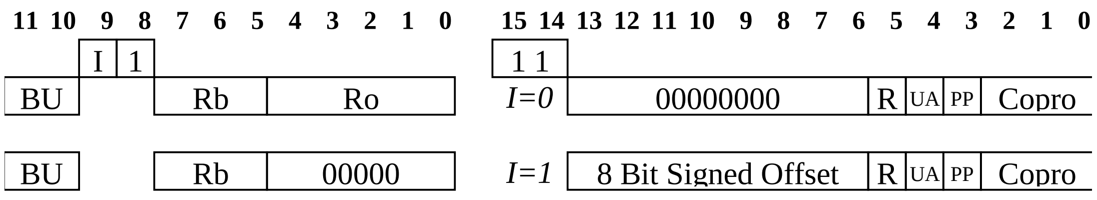
11 10	9	8	7 6 5	4 3 2 1 0
	I	0		
BU			Rs1	Rs2
BU			Rs1	16 Bit Immediate

	15 14	13 12 11 10 9 8 7 6 5 4 3	2 1 0
	1 1		
I=0		0000000000	Copro
I=1		16 Bit Immediate	Copro

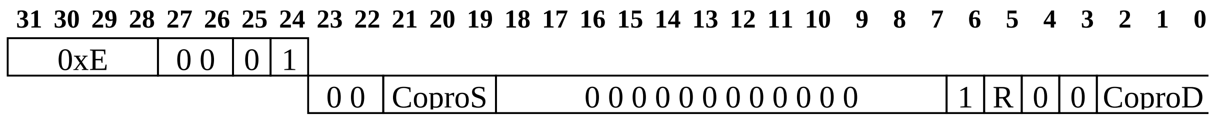
3.6.8. CPWL CP.n,[BBxUaRoPp|BBxUaS8Pp]



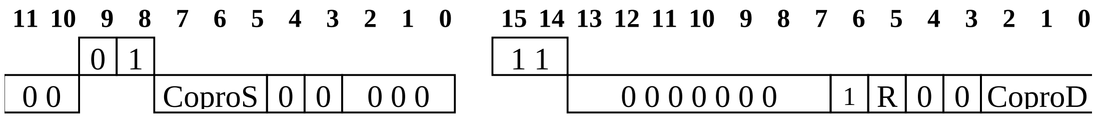
MiniM long encoding



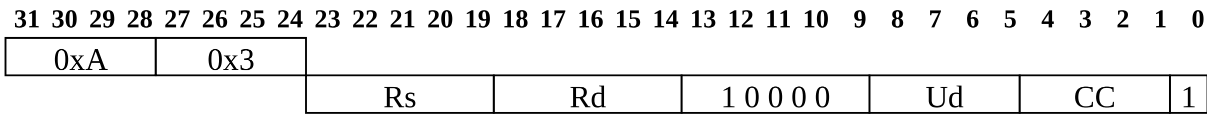
3.6.9. CPXL CP.n,CP.n



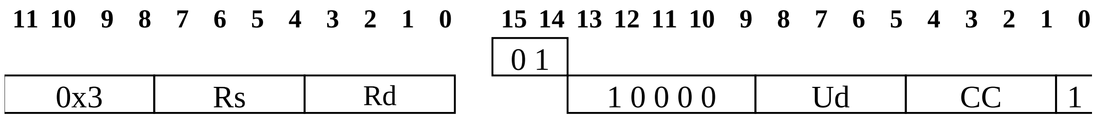
MiniM long encoding



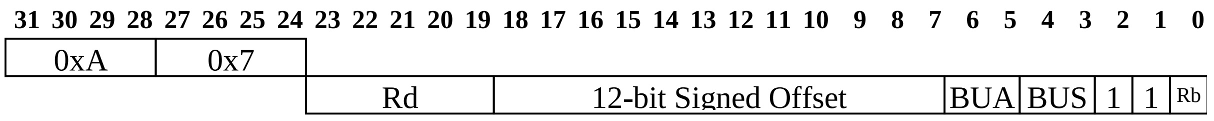
3.6.10. DEFRcc BBx.r,TR.r



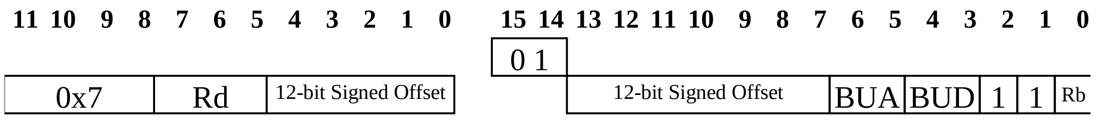
MiniM encoding



3.6.11. GETL BBx.r,BBo.e,[BBa.r+#S12]



MiniM long encoding



3.6.12. GETx BBx.r,[BBa.r+#S12]

31–28	27–24	23–19	18–7	6–5	4–3	2	1	0
0xA	0x7	Rd	12-bit Signed Offset	BUA	BUS	L2	L1	Rb

Note: See section 3.6.11 for 64-bit case (L2=1 and L1=1 not supported by this case).

MiniM long encoding

11–8	7–5	4–0
0x7	Rd	12-bit Signed Offset

15–14	13–7	6–5	4–3	2	1	0
0 1	12-bit Signed Offset	BUA	BUD	L2	L1	Rb

3.6.13. JUMP BBx.r,#X16

31–28	27–24	23–19	18–3	2	1–0
0xA	0xC	Rs	16 Bit Unsigned Immediate	0	BU

MiniM long encoding

11	10	9–5	4–0
0	0	Rs	16 Bit Immediate

15–14	13–3	2	1–0
0 0	16 Bit Immediate	0	BU

3.6.14. KICKcc Ux.r,TR.r

31–28	27–24	23–19	18–14	13–9	8–5	4–1	0
0xA	0x3	Rs	Rd	00000	Ud	CC	1

MiniM long encoding

11–8	7–4	3–0
0x3	Rs	Rd

15–14	13–9	8–5	4–1	0
0 1	0 0 0 0 0	Ud	CC	1

3.6.15. LNKGETL BBx.r,BBo.e,[BBa.r]

31–28	27–24	23–19	18–14	13–8	7	6–5	4–3	2	1	0
0xA	0xD	Rd	Rb	0 0 0 0 0 0	1	BUA	BUR	1	1	0

MiniM long encoding

11–8	7–3	2–0
0x3	Rd	Rb

15–14	13–8	7	6–5	4–3	2	1	0
0 1	0 0 0 0 0 0	1	BUA	BUR	1	1	0

3.6.16. LNKGETx BBx.r,[BBa.r]

31–28	27–24	23–19	18–14	13–8	7	6–5	4–3	2	1	0
0xA	0xD									
		Rs	Rb	0 0 0 0 0 0	1	BUA	BUR	L2	L1	0

Note: See section 3.6.15 for 64-bit case (L2=1 and L1=1 not supported by this case)

MiniM long encoding

11–8	7–3	2–0	15–14	13–8	7	6–5	4–3	2	1	0
			0 1							
0x3	Rd	Rb		0 0 0 0 0 0	1	BUA	BUR	L2	L1	0

3.6.17. LNKSETLcc [BBa.r],BBx.r,BBo.e

31–28	27–24	23–19	18–14	13–10	9	8–7	6–5	4–1	0
0xA	0x4								
Us=Ax|Dx		Rs	Rb	Us	1	0 1	BU	CC	1

MiniM long encoding

11–8	7–3	2–0	15–14	13–10	9	8–7	6–5	4–1	0
			0 1						
0x4	Rs	Rb		Us	1	0 1	BU	CC	1

3.6.18. LNKSETxcc [BBa.r],Ux.r

31–28	27–24	23–19	18–14	13–10	9	8–7	6–5	4–1	0
0xA	0x4								
		Rs	Rb	Us	L2	0 1	BU	CC	L1

Note: See section 3.6.17 for 64-bit case (L2=1 and L1=1 not supported by this case)

MiniM long encoding

11–8	7–3	2–0	15–14	13–10	9	8–7	6–5	4–1	0
			0 1						
0x4	Rs	Rb		Us	L2	0 1	BU	CC	L1

3.6.19. MOV CT.r,#I16

31–28	27–24	23–19	18–3	2	1	0
0xA	0x9					
		Rd	16 Bit Immediate	0	SE	H

MiniM long encoding

11–8	7–5	4–0	15–14	13–3	2	1	0
			0 1				
0x9	Rd	16-bit Immediate		16-bit Immediate	0	SE	H

3.6.20. MOVx RAxx,[BBa.r+#S12]

31 30 29 28	27 26 25 24	23 22 21 20 19	18 17 16 15 14 13 12 11 10 9 8 7	6 5	4 3	2	1	0
0xA	0x6							
		RAxx	12-bit Signed Offset	BUA	00	L2	L1	Rb

MiniM long encoding

11 10 9 8	7 6 5 4 3	2 1 0
0x6	RAxx	Offset 11:9

15 14	13 12 11 10 9 8 7 6 5	4 3	2	1	0
0 1					
	12-bit Signed Offset	BUA	L2	L1	Rb

3.6.21. MOVLcc BBx.r,BBo.r,RD

31 30 29 28	27 26 25 24	23 22 21 20 19	18 17 16 15 14	13 12 11 10 9	8 7	6 5	4 3 2 1	0
0xA	0x2							
		Xs	Rd1	Rd2	00	BU	CC	0

MiniM long encoding

11 10 9 8	7 6 5 4 3	2 1 0
0x2	Xs	Rd1

15 14	13 12	11 10 9	8 7	6 5	4 3 2 1	0
0 1						
	00	Rd2	00	BU	CC	0

3.6.22. MOVxcc Ux.r,RD

31 30 29 28	27 26 25 24	23 22 21 20 19	18 17 16 15 14	13 12 11 10	9	8 7 6 5	4 3 2 1	0
0xA	0x1							
		Xs	Rd	0000	L2	Ud	CC	L1

Note: See section 3.6.21 for 64-bit case (L2=1 and L1=1 not supported by this case)

MiniM long encoding

11 10 9 8	7 6 5 4	3 2 1 0
0x1	Xs	Rd

15 14	13 12 11 10	9	8 7 6 5	4 3 2 1	0
0 1					
	0000	L2	Ud	CC	L1

3.6.23. MOVcc Ux.r,Uy.r

31 30 29 28	27 26 25 24	23 22 21 20 19	18 17 16 15 14	13 12 11 10	9	8 7 6 5	4 3 2 1	0
0xA	0x3							
CC != A		Rs	Rd	Us	0	Ud	CC	0

MiniM long encoding

11 10 9 8	7 6 5 4	3 2 1 0
0x3	Rs	Rd

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

3.6.24. RTDW De.r,Dx.r

31 30 29 28	27 26 25 24	23 22 21 20 19	18 17 16 15 14	13 12 11 10 9 8 7 6 5 4	3	2 1	0
0xA	0xA						
		Rd	Rs	0000000000	S	1 1	DU

MiniM long encoding

11	10	9	8	7	6	5	4	3	2	1	0
0xA				Rd					Rs		

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0 1															
		0000000000										S	1 1		DU

3.6.25. SETL [BBa.r+#S12],BBx.r,BBo.e

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0xA				0x5																											
								Rs					12-bit Signed Offset												BUA		BUS		1	1	Rb

MiniM long encoding

11	10	9	8	7	6	5	4	3	2	1	0
0x5				Rs			12-bit Signed Offset				

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0 1															
		12-bit Signed Offset							BUA		BUS		1	1	Rb

3.6.26. SETx [BBa.r+#S12],BBx.r

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0xA				0x5																											
								Rs					12-bit Signed Offset												BUA		BUS		L2	L1	Rb

Note: See section 3.6.25 for 64-bit case (L2=1 and L1=1 not supported by this case)

MiniM long encoding

11	10	9	8	7	6	5	4	3	2	1	0
0x7				Rd			12-bit Signed Offset				

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0 1															
		12-bit Signed Offset							BUA		BUD		L2	L1	Rb

3.6.27. SETLcc [BBa.r],BBx.r,BBo.e

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0xA				0x4																											
Us=Ax|Dx								Rs					Rb					Us				1	00		BU		CC				1

MiniM long encoding

11	10	9	8	7	6	5	4	3	2	1	0
0x4				Rs					Rb		

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0 1															
		Us				1	00		BU		CC				1

3.6.28. SETXcc [BBa.r],Ux.r

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0xA				0x4																											
								Rs					Rb					Us				L2	00		BU		CC				L1

Note: See section 3.6.27 for 64-bit case (L2=1 and L1=1 not supported by this case)

MiniM long encoding

11 10 9 8 7 6 5 4 3 2 1 0

0x4	Rs	Rb

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

0 1						
	Us	L2	00	BU	CC	L1

3.6.29. SWAPcc Ux.r,Uy.r

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

0xA	0x3							
CC != A		Rs	Rd	Us	1	Ud	CC	0

MiniM long encoding

11 10 9 8 7 6 5 4 3 2 1 0

0x3	Rs	Rd

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

0 1					
	Us	1	Ud	CC	0

3.6.30. XFRX [BBxUaRoPp],[BByUaRoPp]

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

0xD	US	UD	L1	PP						
					Rsb	Rso	Rdb	Rdo	BUS	BUD

MiniM long encoding

11 10 9 8 7 6 5 4 3 2 1 0

US	UD	L1	PP	Rsb	Rso

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

1 0				
	Rdb	Rdo	BUS	BUD

3.6.31. XSDB De.r,Dx.r

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

0xA	0xA						
		Rd	Rs	0000000000	S	0 0	DU

MiniM long encoding

11 10 9 8 7 6 5 4 3 2 1 0

0xA	Rd	Rs1

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

0 1				
	0000000000	S	0 0	DU

3.6.32. XSDW De.r,Dx.r

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

0xA	0xA						
		Rd	Rs	0000000000	S	0 1	DU

MiniM long encoding

11 10 9 8 7 6 5 4 3 2 1 0

0xA	Rd	Rs1

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

0 1				
	0000000000	S	0 1	DU

3.7. Long cache operations

3.7.1. CACHERL BBx.r,BBo.e,[BBa.r+#S6]

31 30 29 28	27 26 25 24	23 22 21 20 19	18 17 16 15 14	13 12 11 10 9 8	7	6 5	4 3	2 1 0
0xA	0xD	Rd	Rb	6-bit Signed Imm	1	BUA	BUR	0 1 1

MiniM long encoding

11 10 9 8	7 6 5 4 3	2 1 0
0xD	Rd	Rb

15 14	13 12 11 10 9 8	7	6 5	4 3	2 1 0
0 1	S6	1	BUA	BUR	0 1 1

3.7.2. CACHERD BBx.r,[BBa.r+#S6]

31 30 29 28	27 26 25 24	23 22 21 20 19	18 17 16 15 14	13 12 11 10 9 8	7	6 5	4 3	2 1 0
0xA	0xD	Rd	Rb	6-bit Signed Imm	1	BUA	BUR	0 0 1

MiniM long encoding

11 10 9 8	7 6 5 4 3	2 1 0
0xD	Rd	Rb

15 14	13 12 11 10 9 8	7	6 5	4 3	2 1 0
0 1	S6	1	BUA	BUR	0 0 1

3.7.3. CACHEWL [BBa.r+#S6],BBx.r,BBo.e

31 30 29 28	27 26 25 24	23 22 21 20 19	18 17 16 15 14	13 12 11 10 9 8	7	6 5	4 3	2 1 0
0xA	0xD	Rs	Rb	6-bit Signed Imm	0	BUA	BUR	0 1 1

MiniM long encoding

11 10 9 8	7 6 5 4 3	2 1 0
0xD	Rs	Rb

15 14	13 12 11 10 9 8	7	6 5	4 3	2 1 0
0 1	S6	0	BUA	BUR	0 1 1

3.7.4. CACHEWD [BBa.r+#S6],BBx.r

31 30 29 28	27 26 25 24	23 22 21 20 19	18 17 16 15 14	13 12 11 10 9 8	7	6 5	4 3	2 1 0
0xA	0xD	Rs	Rb	6-bit Signed Imm	0	BUA	BUR	0 0 1

MiniM long encoding

11 10 9 8	7 6 5 4 3	2 1 0
0xD	Rs	Rb

15 14	13 12 11 10 9 8	7	6 5	4 3	2 1 0
0 1	S6	0	BUA	BUR	0 0 1

3.7.5. DCACHE [BBa.r+#S6],BBx.r

31–28	27–24	23–19	18–14	13–8	7	6–5	4–3	2–0
0xA	0xD	Rs	Rb	6-bit Signed Imm	0	BUA	BUR	000

MiniM long encoding

11–8	7–3	2–0
0xD	Rs	Rb

15–14	13–8	7	6–5	4–3	2–0
0 1	S6	0	BUA	BUR	0 0 0

3.7.6. ICACHE #S15,#S4

31–28	27–24	23–9	8–5	4–1	0
0xA	0xE	15 Bit Signed Offset	0000	PFCount	P

MiniM long encoding

11–8	7–0
0xE	S15

15–14	13–7	6–5	4–1	0
0 1	S15	0 0	PFCount	P

3.7.7. ICACHEEXIT

31–28	27–24	23–9	8–5	4–1	0
0xA	0xE	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0	0 0 0 0	P

MiniM long encoding

11–8	7–0
0xE	0 0 0 0 0 0 0 0

15–14	13–7	6–5	4–1	0
0 1	0 0 0 0 0 0 0	0 0	0 0 0 0	P

3.8. Long trace operations

3.8.1. MOV TT.r,#I16

31–28	27–24	23–19	18–3	2	1	0
0xA	0x9	Rd	16 Bit Immediate	0	SE	H

MiniM long encoding

11–8	7–5	4–0
0x9	Rd	16-bit Immediate

15–14	13–3	2	1	0
0 1	16-bit Immediate	1	SE	H

3.8.2. TTMOV Ux.r,Uy.r

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0xA				0x3																											
								Rs					Rd					Us				1	Ud				CC				1

MiniM long encoding

11	10	9	8	7	6	5	4	3	2	1	0
0x3				Rs				Rd			

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

3.8.3. MOVLcc TTREC,BBx.r,BBo.r

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0xA				0x2																											
								Rs1					Rs2					1 0 0 0 0					BU		00		CC				1

MiniM long encoding

11	10	9	8	7	6	5	4	3	2	1	0
0x2				Rs1				Rs2			

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0 1															
		1 0 0 0 0					BU		00		CC				1

3.9. Shadow Register Access Instruction Coding

3.9.1. MOVRcc Dx.r,AxIReg

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0xA				0x3																											
								0 0 0 0 0					Rd					0	0	0	1	0	0	0	du	au	CC				1

MiniM long encoding

11	10	9	8	7	6	5	4	3	2	1	0
0x3				0 0 0 0				Rd			

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

3.9.2. MOVRcc AxIReg, Dx.r

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0xA				0x3																											
								Rs					0 0 0 0 0					0	0	1	1	0	0	0	au	du	CC				1

MiniM long encoding

11	10	9	8	7	6	5	4	3	2	1	0
0x3				Rs				0 0 0 0			

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

3.9.3. SWAPRcc Dx.r,AxIReg

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0xA				0x3																											
								0 0 0 0 0					Rd					1	0	0	1	0	0	0	du	au	CC				1

MiniM long encoding

11	10	9	8	7	6	5	4	3	2	1	0
0x3				0 0 0 0				Rd			

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

3.9.4. SWAPRcc AxlReg, Dx.r

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0xA				0x3																											
								Rs					0 0 0 0 0					1	0	1	1	0	0	0	au	du	CC				1

MiniM long encoding

11	10	9	8	7	6	5	4	3	2	1	0
0x3				Rs				0 0 0 0			

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

Appendix A. Instruction operation syntax

This section provides an overview of the syntax used in the TRM documents to summarise the execution features of all instructions. This syntax is currently in use in the GP and DSP TRM.

In general, the statements assist analysis of the latency effects of instructions which is hard to describe accurately otherwise.

A.1. _statement_

An instructions behaviour is described via a list of statements. Statements either start with a label or not and consist of a structured comma separated list of individual effects ending in a semi-colon.

Statements without a label at the start describe ‘pre-processing’ steps that aid in the description of the instruction and don’t take ‘real’ time/cycles for the processor to execute:

```
            _effect_, _effect_,
                    … any number of _effect_ …
                         _effect_;
```

Statements with a label execute on the processor and all effects listed get issued in parallel at the time indicated by the label:

```
_label_:    _effect_, _effect_,
                    … any number of _effect_ …
                         _effect_;
```

A.1.1. _label_

A label is one of the following values at the start of a statement followed by a colon:

-1	Indicates a ‘prior dependency’ on inputs/activity in the previous cycle. For example for access to some data/address unit registers there is a stall if the register targeted was updated ‘in unit’ on the previous cycle.
0	Indicates the statement occurs at ‘time zero’; i.e. in the same cycle in which the instruction is issued. If an instruction has no effects after time zero then all its results are available immediately for use by a subsequent instruction unless that instruction indicates that it has an explicit ‘prior dependency’ (see previous).
1 2 3 …	Indicates that the statement is executed the specific number of cycles after ‘time zero’. The results of the statement are only available to further instructions after the time stated.
n	This describes a multi-issue pattern where ‘time zero’ is repeated a number of times in order to complete the execution. The repeat count n is generally obvious from the nature of the instruction concerned.
n-1 n+1 n+2 n+3	These expressions describe ‘prior dependency’ or ‘latency’ relative to a multi-issue ‘time zero’ event.

2n	This scaled multi-issue pattern implies that twice as many issues are needed than may be expected (e.g. a 32-bit core executing a 64-bit operation). Constant positive/negative offset values will be scaled up to match so that the expression never gets more complex than 2n-p or 2n+d.

A.2. _effect_

Effects are generally assignments of a value to a location followed comma or semi-colon. Three types of effect syntax are used starting with the simple effects:

```
        _location_ = _value_
```

Or compound effects with corresponding comma separated lists each side of the assignment:

```
        _location_, _location_ …      = _value_, _value_ …
```

Note: Commas are used for both this syntax element and the top level list of effects within a statement. It's important to count the commas before the assignment operator in order to work out where the compound effect ends and the enclosing statement continues.

Or conditional effects (which can also be expanded to use the compound case):

```
        _condition_ -> _location_ = _value_
```

Or the invocation of a procedure:

```
        _function_call_
```

A.2.1. _condition_

A condition specifies a true/false value and can consist of a simple instruction modifier or option:

```
        [Z]
        <Q>
```

An expression using C logical operation syntax delineated by round brackets:

```
        ( … && … || … ! … )
```

Each of the elements of a conditional expression may be conditions or values with 'not equal to zero' being the assumed conversion between simple value and true/false expressions.

More round brackets may be used within the conditional expression to make the meaning clear, however the outer set of round brackets is required even if the expression could be parsed in an unambiguous manner without them; e.g. (! [Z]).

A.3. _location_

A location specifies one of the following:

Reg	A register (typically abstracted) that exists in the processor; eg. De.r or TXMODE.
Reg.Field	A register field that exists in the processor; e.g. TXMODE.DUAccSat.
Reg.o	A register in the opposite matched data or address unit to that specified abstractly in the Register; e.g. De.o.

Reg.b Reg.t	The top most-significant or bottom least-significant half of a register.
Reg.ob Reg.ot	A combination of the previous two notations.
Variable	An invented scratch location that can be setup and then accessed in an arbitrary but essentially sequential manner by the direct statements or procedures that make up an instruction description. For example, DUFlags is a hidden secondary result of many procedures that permits an instruction description to optionally indicate if it modifies the current flag state.

Also any location expression can be preceded by a special apostrophe character that denotes "new value of" the location concerned so that new and old values of any location may be referenced:

```
        '_location_
```

Note: Memory operations are all described via function or procedure calls so that side effects of addressing can be handled easily .

A.4. _value_

A value is typically one of the following:

```
        _location_
        _function_call_
        _condition_
        _number_
```

Or an expression using C expression with round brackets deployed internally:

```
        … + ( … - … ) / …
```

Note: To differentiate true/false _condition_ and these more general value expressions there are no round brackets around this type of expression.

Or a compound value which implies concatenation of data into a structure-like data record:

```
        { _value_, _value_, … }
```

A.4.1. _number_

Simple decimal or hexadecimal values. The 0x notation is used for hex values with unary minus/plus/invert supported as required using C syntax.

A.5. _function_

The general syntax of a procedure call is similar whether it returns a value or not:

```
        Function( _location_, ..., _value_, ... )
```

Functions are arbitrary named elements that are then described elsewhere in the document concerned. Functions operate entirely in a manner consistent with textual expansion within any context they are referenced. We don't support recursion or pointers.

Note: Because of the equivalence to textual expansion a function can access both the old and new value of any _location_ passed into it.

Appendix B. Alphabetical index of instructions

Instruction	Section
CMPcc Dx.r,De.r	2.3.16
CMPcc Dx.r,Rx.r	2.3.17
CPRLcc BBx.r,BBo.r,CP.n	2.6.3
CPRR [BBxUaRoPp],CP.r	2.6.4
CPRx [BBxUaRoPp|BBxUaS6Pp],CP.n	2.3.19
CPRxcc [BBx.r],CP.n	2.6.6
CPRxcc Ux.r,CP.n	2.6.5
CPWL CP.n,[BBxUaRoPp|BBxUaS8Pp]	2.6.8
CPWL CP.n,BBx.r,BBo.r|#X16	2.6.7
CPXL CP.n,CP.n	2.6.9
DCACHE [BBa.r+#S6],BBx.r	2.7.5
DEFRcc BBx.r,TR.r	2.6.10
FFB De.r,Dx.r	2.4.2
GETD BBx.r, [A0.r+#S3]	2.2.13
GETL BBx.r,BBo.e,[BBa.r+#S12]	2.6.11
GETL BBx.r,BBo.e,[BBxUaRoPp|BBxUaS6Pp]	2.3.20
GETx BBx.r,[BBa.r+#S12]	2.6.12
GETx Ux.r,[BBxUaRoPp|BBxUaS6Pp]	2.3.21
ICACHE #S15,#S4	2.7.6
ICACHEEXIT	2.7.7
JUMP BBx.r,#X16	2.6.13
KICKcc Ux.r,TR.r	2.6.14
LNKGETL BBx.r,BBo.e,[BBa.r]	2.6.15
LNKGETx BBx.r,[BBa.r]	2.6.16
LNKSETLcc [BBa.r],BBx.r,BBo.e	2.6.17
LNKSETxcc [BBa.r],Ux.r	2.6.18
LOCKn	2.2.14
MAX De.r,Dx.r,De.r	2.4.3
MDRD RD..RD..	2.3.22
MGET BBx.r..BBe.r..,[BBa.r]	2.2.15
MIN De.r,Dx.r,De.r	2.4.4
MMOVx BBx.r..BBe.e..,RD	2.3.23
MMOVx RAxx..RAee..,[BBx.r]	2.3.24
MORT De.r,Dx.r,De.r	2.4.5
MORT De.r,Dx.r,Rx.r	2.4.6
MOV Ae.r,Ax.r	2.2.16
MOV Ae.r,Rx.r	2.3.25

Instruction	Section
MOV Ax.r,#I16	2.2.17
MOV CT.r,#I16	2.6.19
MOV De.r,Dx.r	2.2.18
MOV Dx.r,#I16	2.2.19
MOV Dx.r,Rx.r	2.3.26
MOV TT.r,#I16	2.8.1
MOV Ux.r,Uy.r	2.2.20
MOVcc Ux.r,Uy.r	2.6.23
MOVLcc BBx.r,BBo.r,RD	2.6.21
MOVLcc TTREC,BBx.r,BBo.r	2.8.3
MOVTT Ux.r,Uy.r	2.8.2
MOVx RAxx,[BBa.r+#S12]	2.6.20
MOVx RAxx,[BBxUaRoPp|BBxUaS6Pp]	2.3.27
MOVxcc Ux.r,RD	2.6.22
MSETx [BBa.r],BBx.r..BBe.r..	2.2.21
MUL BBx.r,Dx.r,#X8	2.3.31
MUL BBx.r,Dx.r,De.r	2.3.29
MUL BBx.r,Dx.r,Rx.r	2.3.30
MUL De.r,Dx.r,#I16	2.2.23
MUL De.r,Dx.r,De.r	2.2.22
MUL De.r,Dx.r,Rx.r	2.3.28
MULcc De.r,Dx.r,#X8	2.3.34
MULcc De.r,Dx.r,De.r	2.3.32
MULcc De.r,Dx.r,Rx.r	2.3.33
NEG Ae.r,Ax.r	2.2.24
NEG Ae.r,Rx.r	2.3.35
NEG Ax.r,#I16	2.2.25
NEG De.r,#I16	2.2.27
NEG De.r,Dx.r	2.2.26
NEG De.r,Rx.r	2.3.36
NMIN De.r,Dx.r,De.r	2.4.7
NOP	2.2.28
NORM De.r,Dx.r	2.4.8
OR De.r,Dx.r,#I16	2.2.30
OR De.r,Dx.r,De.r	2.2.29
OR De.r,Dx.r,Rx.r	2.3.37
OR Rx.r,Dx.r,#X8	2.3.40

Instruction	Section
ORcc De.r,Dx.r,#X8	2.3.41
ORcc Rx.r,Dx.r,De.r	2.3.38
ORcc Rx.r,Dx.r,Rx.r	2.3.39
RTDW De.r,Dx.r	2.6.24
RTH	2.2.31
RTI	2.2.32
SETD [A0.r+#S3],BBx.r	2.2.33
SETL [BBa.r+#S12],BBx.r,BBo.e	2.6.25
SETL [BBxUaRoPp|BBxUaS6Pp],BBx.r,BBo.e	2.3.42
SETLcc [BBa.r],BBx.r,BBo.e	2.6.27
SETx [BBa.r+#S12],BBx.r	2.6.26
SETx [BBxUaRoPp|BBxUaS6Pp],Ux.r	2.3.43
SETXcc [BBa.r],Ux.r	2.6.28
SHIFT De.r,Dx.r,De.r|#X5	2.5.1
SHIFT Rx.r,Dx.r,De.r|#X5	2.5.2
SHIFTcc De.r,Dx.r,De.r|#X5	2.5.3
SUB Ae.e,Ax.r,#I16	2.2.35
SUB Ae.r,Ax.r,Ae.r	2.2.34
SUB Ae.r,Ax.r,Rx.r	2.3.44
SUB De.e,Dx.r,#I16	2.2.37
SUB De.r,Dx.r,De.r	2.2.36
SUB De.r,Dx.r,Rx.r	2.3.49
SUB Rx.r,Ax.r,#X8	2.3.47
SUB Rx.r,Dx.r,#X8	2.3.52
SUBcc Ae.r,Ax.r,#X8	2.3.48
SUBcc De.r,Dx.r,#X8	2.3.33
SUBcc Rx.r,Ax.r,Ae.r	2.3.45
SUBcc Rx.r,Ax.r,Rx.r	2.3.46
SUBcc Rx.r,Dx.r,De.r	2.3.50
SUBcc Rx.r,Dx.r,Rx.r	2.3.51
SWAP Ux.r,Uy.r	2.2.28
SWAPcc Ux.r,Uy.r	2.6.29
SWITCH #-1|0xC3020E|0xFF3FFF	2.2.39
TST Dx.r, #I16	2.2.41
TST Dx.r,De.r	2.2.40
TST Dx.r,Rx.r	2.2.42
TSTcc Dx.r,#X8	2.3.56

Instruction	Section
TSTcc Dx.r,De.r	2.3.54
TSTcc Dx.r,Rx.r	2.3.55
VPACK De.r,Dx.r,De.r	2.4.9
VPACK De.r,Dx.r,Rx.r	2.4.10
XFRX [BBxUaRoPp],[BByUaRoPp]	2.6.30
XOR De.e,Dx.r,#I16	2.2.44
XOR De.r,Dx.r,De.r	2.2.43
XOR De.r,Dx.r,Rx.r	2.3.57
XOR Rx.r,Dx.r,#X8	2.3.60
XORcc De.r,Dx.r,#X8	2.3.61
XORcc Rx.r,Dx.r,De.r	2.3.58
XORcc Rx.r,Dx.r,Rx.r	2.3.59
XSDB De.r,Dx.r	2.6.31
XSDW De.r,Dx.r	2.6.32