Optimum Semiconductor Technologies
&
General Processor Technologies

Unity Instruction Set Architecture Manual

Document Number: GPT-ISA-DSP

Date Released: Mar 27th, 2019

Designer Approval: _____

Reviewer Approval: _____

Issued by:
Optimum Semiconductor Technologies, Inc.
120 White Plains Road, 4th Floor
Tarrytown, NY 10591**,** USA
Phone: +1 (914) 287-8500
Fax: +1 (914) 287-8501

Document Revision History

Revision	Section	Description	Date
0.00	All	Initial release	05-27-2019

Document Conventions

Table of Contents

List of Figures

No table of figures entries found.

List of Tables

No table of figures entries found.

1 Introduction

1.1 Processor Overview

The Unity architecture consists of a 32-bit scalar processing core, with vector extension. The architected state that comprises the core includes:

- 16 32-bit general-purpose registers (GPR), written as `$r0`…`$r15`.
- A carry/saturate bit.
- A 2-bit FIXCR register
- 16 32-bit address registers (ADR), written as `$a0`…`$a15`
- 2 branch target registers, a branch intermediate instruction and a branch return register, all of which are 32-bit, written as `$t0`, `$t1`, `$t4` and `$t7`.

There is a set of instructions to manipulate the values in the GPRs. A smaller set operates on the ADRs and BTRs. For more extensive operation the values in ADR/BTR can be transferred to the GPR, modified, and copied back.

All communication with external memory is done via explicit loads and store instructions. The address accessed is read from an address register or address register offset.

All program control transfers are to targets stored in a branch register or to targets computed by register and immediate. A conditional jump is accomplished by instructions that specify the condition to be evaluated, and a branch target register which contains the PC that should be executed if the condition is true.

The vector unit is based on variable-length vectors, operating on vector registers[1], with instructions picked to supporting digital signal processing. It adds:

- 16 128-bit vector registers, `$v0`...`$v15`

The non-user state of the processor is implementation defined. It can be accessed by a user via calls into a hardware abstraction layer[2]. This layer is also used to implement complex operations such as atomic memory accesses.

1.2 Conventions

1.2.1 Undefined behavior

At various places in this document, instructions are specified as having undefined behavior in some scenarios. If such a scenario occurs, different implementations can have different behavior. This behavior can include

- Continuing silently, with some possibly non-repeatable result
- Raising a recoverable internal exception
- Entering some unrecoverable state

1.2.2 Unspecified fields

Certain bit-ranges in some instructions (generally written as /// in the instruction description) are unused by the instruction. Unless otherwise specified, these must be written with 0 – any other value will result in undefined behavior.

1.2.3 Constant fields

Certain instructions have constant fields that feed the operation, such as the 5-bit shift value in a shift immediate. Other fields modify the behavior of the instruction, such as the MD sign type field of a `max` instruction, which identifies whether it is a signed or unsigned value compare. In the assembler, immediate values will be written as an operand of the instruction, so it will be `shli $r3,$r7,5`, while instruction modifiers will be written as extensions to the operator, so it will be `max_s` for signed value compare.

In general, we have not yet documented all legal instruction modification forms.

[2] This is similar to the PALcode of the Alpha architecture.

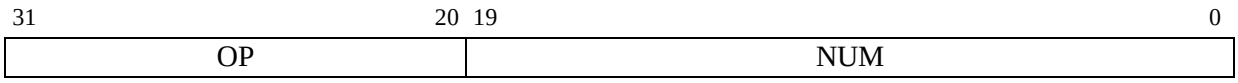
1.3 Instruction Formats

Instructions are 4 bytes long and aligned on 4 byte boundaries. Bits 31:20 are used to specify the operation (OP). Other fields in the operation can specify:

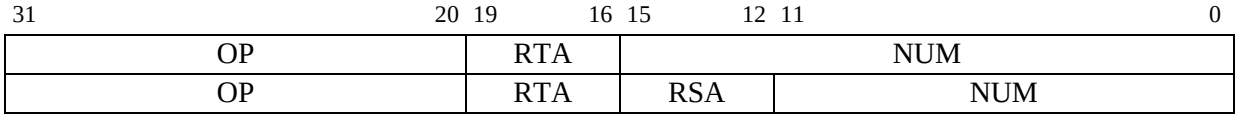
- The registers used in the instruction
- Signed or unsigned immediate values
- Modifications on the behavior of the instruction

The format diagrams below show all valid combinations of instruction fields. Undefined instructions (shown as ///) *must* be set to 0, unless otherwise documented.

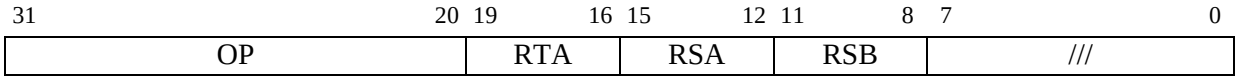
1.3.1 NUM format



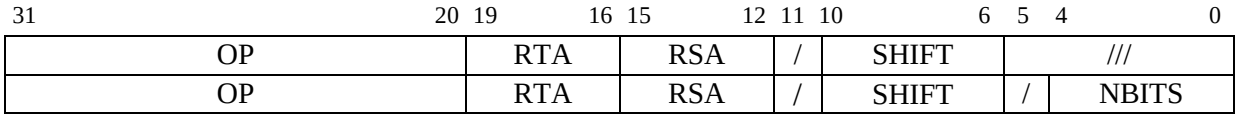
1.3.2 RNUM format



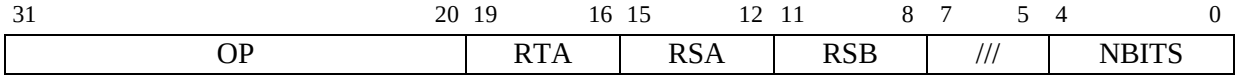
1.3.3 RR format



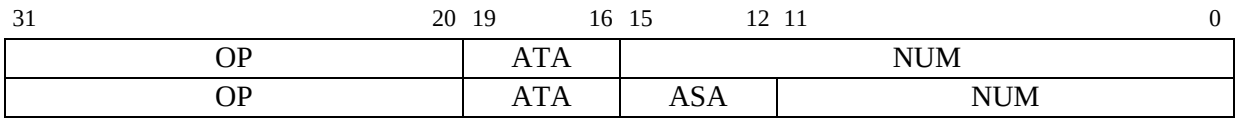
1.3.4 RSH format



1.3.5 RRNB format



1.3.6 ANUM format



1.4 Instruction Fields

The ISA defines the following instruction fields:

ASA(15:12) Specifies an ADR to be used as a source

ATA(19:16) Specifies an ADR to be used as a target

NBITS(4:0) Immediate, specifies number of bits to extract

NUM(11/15/19:0) Signed immediate, provides an input to the operation

RSA(15:12) Specifies a GPR to be used as a source

RSB(11:8) Specifies a GPR to be used as a source

RTA(19:16) Specifies a GPR to be used as a target

SHIFT(10:6) Immediate, specifies number of bits to shift other input

2 Branch Unit

2.1 Overview

Unity has two type branches: branch to values computed by a 32-bit register and an immediate; branches to values stored in a 32-bit register. In this chapter, we shall describe these registers, and the instructions that use these registers. Note that while these instructions can read the program counter, or modify it, the program counter is not a generally available register.

2.2 Branch Registers

2.2.1 Target Registers

There are two 32 bit jump target registers, `$t0` and `$t1`[3]. They provide the destination address for all jump instructions other than **ret**.

2.2.2 Return Register

There is a single branch return register, It is implicitly set by the **call** instruction, and is the destination of the **ret** instruction. It is written as `$t7` in instructions where it must be explicitly indicated.

2.2.3 Intermediate Register

There is a single branch intermediate register. It can be used as part of a target computation, but cannot be used as the destination of any instruction. It is written as `$t4` in instructions where it must be explicitly indicated.

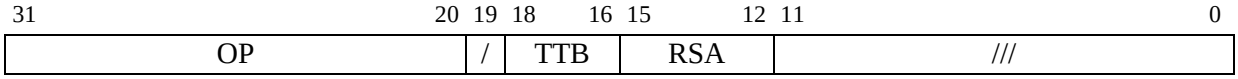
[3] The current encodings reserve space for up to 4 target registers. Future versions of the architecture may choose to make more target registers available.

2.3 Branch Instructions

2.3.1 Branch Target Instructions

2.3.1.1 Set Branch Target from GPR

tsetr $ttb,$rsa



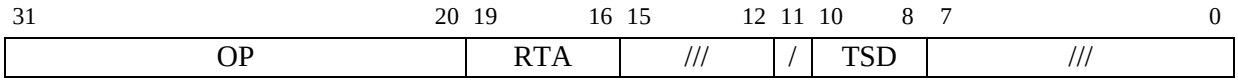
TTB ← [RSA]

Copy the 32-bit value from the general-purpose register RSA to the target branch register TTB.

The results of this operation are undefined if the value stored into the TTB is not 4B aligned. The results of this operation are undefined if TTB is not one of 0, 1, 4 or 7.

2.3.1.2 Set GPR from Branch Target

rsett $rta,$tsd



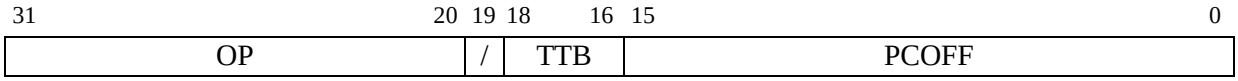
RTA ← [TSD]

Copy the 32-bit value from the branch target register TSD to the general-purpose register RTA.

The results of this operation are undefined if TSD is not one of 0, 1, 4 or 7.

2.3.1.3 Branch Target from PC plus Immediate

taddpci $ttb,pcoff

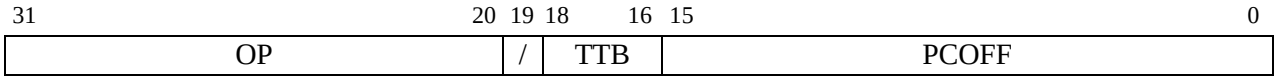


TTB ← PC + PCOFF<<2

Set the value of branch target register TTB to the PC of the instruction plus sign extended immediate PCOFF shifted by 2.

2.3.1.4 Branch Target from PC plus Immediate Long

taddpcil $ttb,pcoff

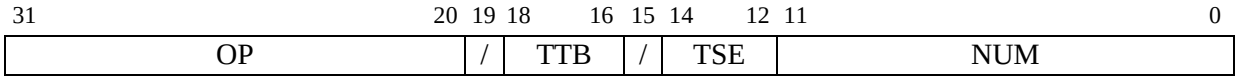


TTB ← (PC$_{31:14}$ + PCOFF<<14).0$_{13:0}$

Set the value of branch target register TTB to the upper 18 bits of the PC of the instruction plus sign extended 16-bit immediate PCOFF shifted by 14; the lower 14 bits are zeroed.

2.3.1.5 Branch Target plus Immediate

taddti $ttb,$tse,num

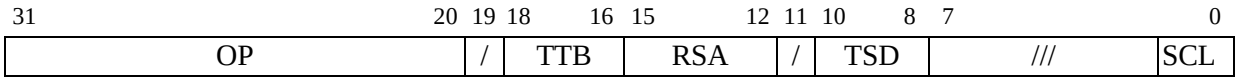


TTB ← TSE + NUM<<2

Set the value of branch target register TTB to the contents of target register TSE plus zero extended 12-bit immediate NUM shifted by 2.

2.3.1.6 Branch Target plus Register

taddtr_scl $ttb,$tsd,$rsa



TTB ← [TSD] + [RSA]<<SCL

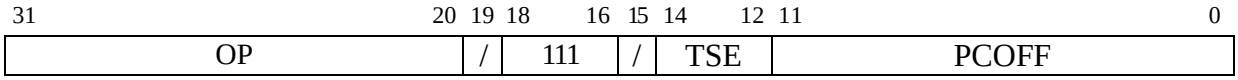
The target register TTB is set to the sum of the contents of target register TSD and general-purpose register RSA shifted by 0/1/2 or 3 bits, based on the SCL field.

In the assembler, the mnemonics `taddtr_b`/ `taddtr_h`/ `taddtr_w` are used for SCL=0/1/2.

2.3.2 Unconditional Jump Instructions

2.3.2.1 Call Branch Target plus Immediate

`callti $tse, pcoff`

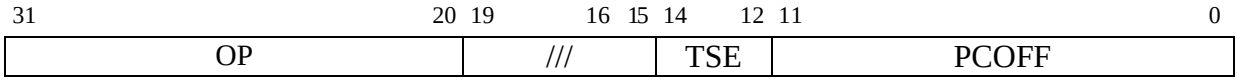



```
T7 ← PC + 4
PC ← TSE + PCOFF<<2
```

Set the value of the program counter to the contents of branch target register TSE plus zero extended immediate PCOFF shifted by 2

2.3.2.2 Jump Branch Target plus Immediate

`jti $tse,pcoff`

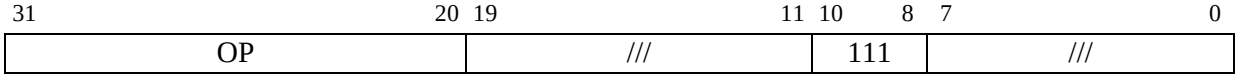



```
PC ← TSE + PCOFF<<2
```

Set the value of the program counter to the contents of TSE plus zero extended immediate PCOFF shifted by 2

2.3.2.3 Return

`ret`

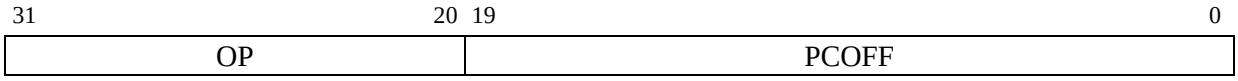



```
PC ← T7
```

Set the program counter to the contents of the branch return register.

2.3.2.4 Direct Jump

dj pcoff

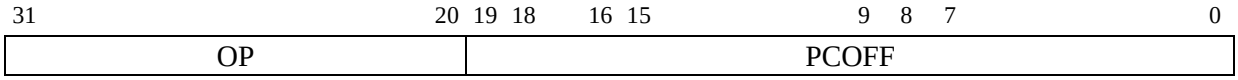



```
PC ← PC + PCOFF<<2
```

Set the value of the program counter to the contents of the instruction PC plus sign-extended immediate PCOFF shift left by 2.

2.3.2.5 Direct Call

dcall pcoff

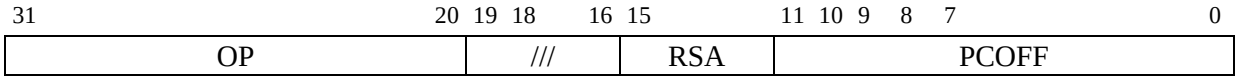



```
T7 ← PC + 4
PC ← PC + PCOFF<<2
```

Copy the return address (PC + 4) to the branch return register. Set the value of the program counter to the contents of the instruction PC plus sign-extended immediate PCOFF shift left by 2.

2.3.2.6 Repeat

repeat $rsa,pcoff




```
Repeat execution [RSA] times of instruction sequence:
From [PC+4] to [PC+PCOFF]
```

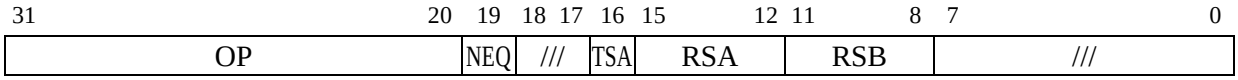
Repeat execution an instruction sequence RSA times. The first instruction address of sequence is PC + 4, the next instruction of REPEAT, the last instruction address is PC + PCOFF. PCOFF is a 12-bit unsigned immediate.

Software should make sure there is no branch and repeat instruction in repeat instruction sequence, otherwise the hardware behavior is unpredictable. Asynchronous interrupt will be disable in repeat sequence execution.Synchronous exception will set processor in unpredictable state.

2.3.3 Conditional Integer Jump Instructions

2.3.3.1 Jump Equal Word

jcw_neq $tsa,$rsa,$rsb




```
if( (NEQ = 1b && ([RSA] = [RSB]))
||  (NEQ = 0b && ([RSA] ≠ [RSB])) )
  PC ←[TSA]
else
  PC ← PC + 4
```

If the branch taken condition is true, i.e.

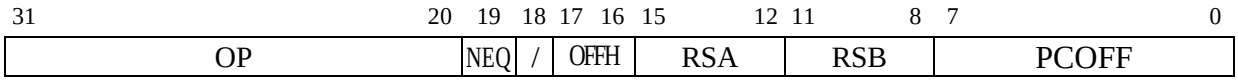
- NEQ is 1 and general-purpose registers RSA and RSB equal each other, or
- NEQ is 0 and general-purpose registers RSA and RSB do not equal each other

set the value of the program counter to the contents of the branch target register TSA.

In the assembler, the mnemonics `jcw_ne`/ `jcw_eq` are used for NEQ=0/1

2.3.3.2 Direct Jump Equal Word

djcw_neq pcoff,$rsa,$rsb




```
if( (NEQ = 1b && ([RSA] = [RSB]))
||  (NEQ = 0b && ([RSA] ≠ [RSB])) )
  PC ← PC + ( OFFH.PCOFF ) << 2
else
  PC ← PC + 4
```

If the branch taken condition is true, i.e.

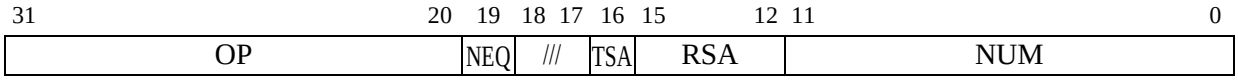
- NEQ is 1 and general-purpose registers RSA and RSB equal each other, or
- NEQ is 0 and general-purpose registers RSA and RSB do not equal each other

set the value of the program counter to the sum of program counter plus sign-extended immediate OFFH connect PCOFF shifted by 2.

In the assembler, the mnemonics `djcw_ne`/ `djcw_eq` are used for NEQ=0/1

2.3.3.3 Jump Equal Word Immediate

`jcwi_neq $tsa,$rsa,num`




```
if( (EQ = 1b && ([RSA] = NUM ))
||  (EQ = 0b && ([RSA] ≠ NUM )) )
  PC ← [TSA]
else
  PC ← PC + 4
```

If the branch taken condition is true, i.e.

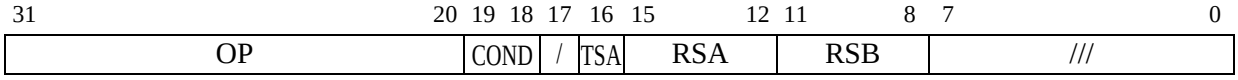
- NEQ is 1 and general-purpose register RSA equals the sign-extended 12-bit immediate value of NUM, or
- NEQ is 0 and general-purpose register RSA does not equal the sign-extended 12-bit immediate value of NUM

set the value of the program counter to the contents of the branch target register TSA.

In the assembler, the mnemonics `jcwi_ne`/ `jcwi_eq` are used for NEQ=0/1

2.3.3.4 Jump Conditional Signed Word

`jcsw_cond $tsa,$rsa,$rsb`




```
if( ((COND = 00b) && [RSA] > [RSB])
||  ((COND = 01b) && [RSA] < [RSB])
||  ((COND = 10b) && [RSA] ≤ [RSB])
||  ((COND = 11b) && [RSA] ≥ [RSB])))
  PC ← [TSA]
else
  PC ← PC + 4
```

The contents of general-purpose registers RSA and RSB are compared to each other as signed values. The COND specifies the test.

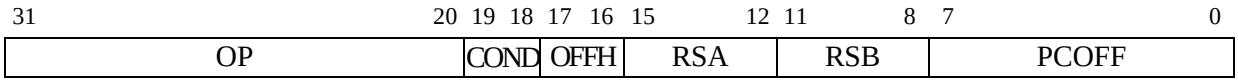
- COND = 0: signed greater than
- COND = 1: signed less than
- COND = 2: signed less than or equal to
- COND = 3: signed greater than or equal to

If the test is satisfied, set the value of the program counter to the contents of the branch target register TSA.

In the assembler, the mnemonics `jcsw_gt/ jcsw_lt/ jcsw_le/ jcsw_ge` are used for COND=0/1/2/3.

2.3.3.5 Direct Jump Conditional Signed Word

djcsw_cond pcoff,$rsa,$rsb




```
if( ((COND = 00b) && [RSA] > [RSB])
||  ((COND = 01b) && [RSA] < [RSB])
||  ((COND = 10b) && [RSA] ≤ [RSB])
||  ((COND = 11b) && [RSA] ≥ [RSB])))
  PC ← PC + ( OFFH.PCOFF) << 2
else
  PC ← PC + 4
```

The contents of general-purpose registers RSA and RSB are compared to each other as signed values. The COND specifies the test.

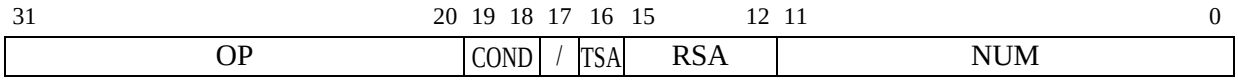
- COND = 0: signed greater than
- COND = 1: signed less than
- COND = 2: signed less than or equal to
- COND = 3: signed greater than or equal to

If the test is satisfied, set the value of the program counter to the sum of program counter plus sign-extended immediate OFFH connect PCOFF shifted by 2.

In the assembler, the mnemonics `djcsw_gt/ djcsw_lt/ djcsw_le/ djcsw_ge` are used for COND=0/1/2/3

2.3.3.6 Jump Conditional Signed Immediate

jcswi_cond $tsa,$rsa,num



```
if( ((COND = 00b) && [RSA] > NUM )
||  ((COND = 01b) && [RSA] < NUM )
||  ((COND = 10b) && [RSA] ≤ NUM )
||  ((COND = 11b) && [RSA] ≥ NUM ) )
  PC ← [TSA]
else
  PC ← PC + 4
```

The contents of general-purpose registers RSA are compared to the sign-extended 12 bits value NUM. The COND specifies the test.

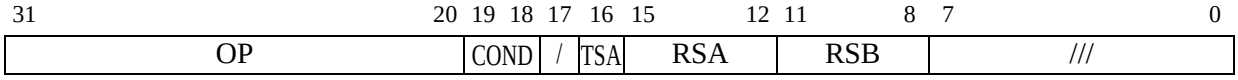
- COND = 0: signed greater than
- COND = 1: signed less than
- COND = 2: signed less than or equal to
- COND = 3: signed greater than or equal to

If the test is satisfied, set the value of the program counter to the contents of the branch target register TSA.

In the assembler, the mnemonics `jcswi_gt`/ `jcswi_lt`/ `jcswi_le`/ `jcswi_ge` are used for COND=0/1/2/3

2.3.3.7 Jump Conditional Unsigned Word

`jcuw_cond $tsa,$rsa,$rsb`




```
if( ((COND = 00b) && [RSA] >u [RSB])
||  ((COND = 01b) && [RSA] <u [RSB])
||  ((COND = 10b) && [RSA] ≤u [RSB])
||  ((COND = 11b) && [RSA] ≥u [RSB]) )
  PC ← [TSA]
else
  PC ← PC + 4
```

The contents of general-purpose registers RSA and RSB are compared to each other as unsigned values. The COND specifies the test.

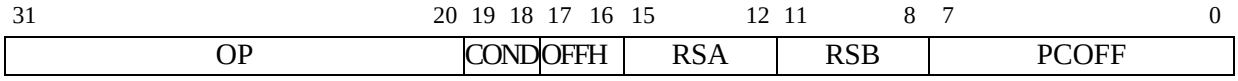
- COND = 0: unsigned greater than
- COND = 1: unsigned less than
- COND = 2: unsigned less than or equal to
- COND = 3: unsigned greater than or equal to

If the test is satisfied, set the value of the program to the contents of the branch target register TSA.

In the assembler, the mnemonics `jcuw_gt/ jcuw_lt/ jcuw_le/ jcuw_ge` are used for COND=0/1/2/3

2.3.3.8 Direct Jump Conditional Unsigned Word

djcuw_cond pcoff,$rsa,$rsb




```
if( ((COND = 00b) && [RSA] >u [RSB])
||  ((COND = 01b) && [RSA] <u [RSB])
||  ((COND = 10b) && [RSA] ≤u [RSB])
||  ((COND = 11b) && [RSA] ≥u [RSB]) )
  PC ← PC + ( OFFH.PCOFF ) << 2
else
  PC ← PC + 4
```

The contents of general-purpose registers RSA and RSB are compared to each other as unsigned values. The COND specifies the test.

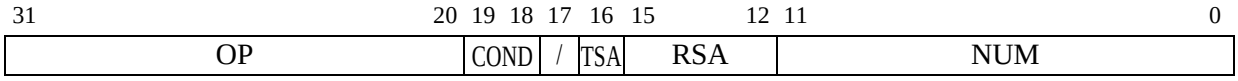
- COND = 0: unsigned greater than
- COND = 1: unsigned less than
- COND = 2: unsigned less than or equal to
- COND = 3: unsigned greater than or equal to

If the test is satisfied, set the value of the program to the sum of program counter of the instruction plus sign-extended immediate OFFH connect PCOFF shifted by 2.

In the assembler, the mnemonics `djcuw_gt/ djcuw_lt/ djcuw_le/ djcuw_ge` are used for COND=0/1/2/3

2.3.3.9 Jump Conditional Unsigned Immediate

`jcuwi_cond $tsa,$rsa,num`




```
if( ((COND = 00b) && [RSA] >u NUM )
||  ((COND = 01b) && [RSA] <u NUM )
||  ((COND = 10b) && [RSA] ≤u NUM )
||  ((COND = 11b) && [RSA] ≥u NUM ) )
  PC ← [TSA]
else
  PC ← PC + 4
```

The contents of general-purpose registers RSA are compared to the **sign-extended** 12 bits value NUM. The COND specifies the test.

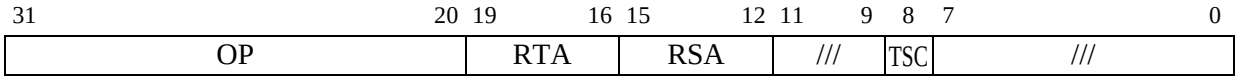
- COND = 0: unsigned greater than
- COND = 1: unsigned less than
- COND = 2: unsigned less than or equal to
- COND = 3: unsigned greater than or equal to

If the test is satisfied, set the value of the program to the contents of the branch target register TSA.

In the assembler, the mnemonics `jcuwi_gt/ jcuwi_lt/ jcuwi_le/ jcwui_ge` are used for COND=0/1/2/3

2.3.3.10 Jump If Decrement Zero

jdecz $tsc,$rta,$rsa

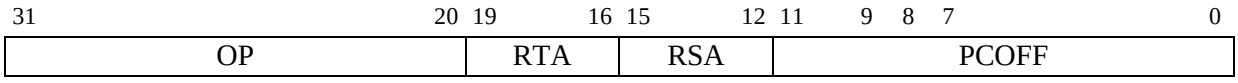



```
If( [RSA] = 1 )
  PC ← [TSC]
else
  PC ← PC + 4
RTA ← [RSA]-1
```

Set the value of the program counter to the contents of the branch target register TSA if the value in RSA is 1. Set RTA to the value of RSA–1.

2.3.3.11 Direct Jump If Decrement Zero

djdecz pcoff,$rta,$rsa

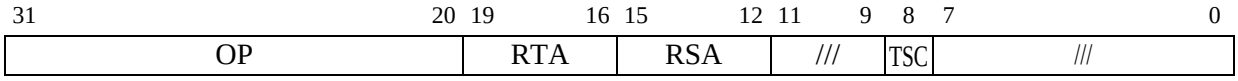



```
If( [RSA] = 1 )
  PC ← PC + PCOFF << 2
else
  PC ← PC + 4
RTA ← [RSA]-1
```

Set the value of the program counter to the sum of program counter of the instruction plus sign-extended immediate PCOFF shifted by 2 if the value in RSA is 1. Set RTA to the value of RSA – 1.

2.3.3.12 Jump If Decrement Non Zero

jdecnz $tsc,$rta,$rsa

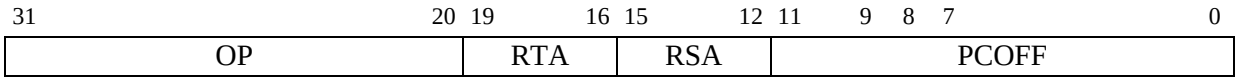



```
If( [RSA] ≠ 1 )
  PC ← [TSC]
else
  PC ← PC + 4
RTA ← [RSA]-1
```

Set the value of the program counter to the contents of the branch target register TSA if the value in RSA is not 1. Set RTA to the value of RSA – 1.

2.3.3.13 Direct Jump If Decrement Non Zero

djdecnz pcoff,$rta,$rsa



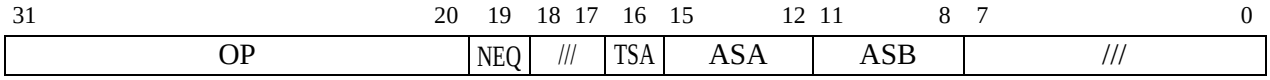

```
If( [RSA] ≠ 1 )
  PC ← PC + PCOFF << 2
else
  PC ← PC + 4
RTA ← [RSA]-1
```

Set the value of the program counter to the sum of program counter of the instruction plus sign-extended immediate PCOFF shifted by 2 if the value in RSA is not 1. Set RTA to the value of RSA – 1.

2.3.4 Conditional Address Jump Instructions

2.3.4.1 Jump Address Equal

`jca_eq $tsa,$asa,$asb`




```
if( (EQ = 1b && ([ASA] = [ASB]))
||  (EQ = 0b && ([ASA] ≠ [ASB])) )
  PC ← [TSA]
else
  PC ← PC + 4
```

If the branch taken condition is true, i.e. if

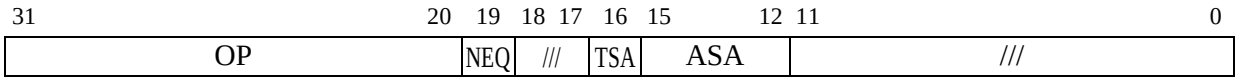
- NEQ is 1 and address registers ASA and ASB equal each other, or
- NEQ is 0 and address registers ASA and ASB do not equal each other

set the value of the program counter to the contents of the branch target register TSA.

In the assembler, the mnemonics `jca_ne`/ `jca_eq` are used for NEQ=0/1

2.3.4.2 Jump Address Equal Zero

`jcaz_neq $tsa,$asa`




```
if( (EQ = 1b && ([ASA] = 0))
||  (EQ = 0b && ([ASA] ≠ 0)) )
  PC ← [TSA]
else
  PC ← PC + 4
```

If the branch taken condition is true, i.e. if

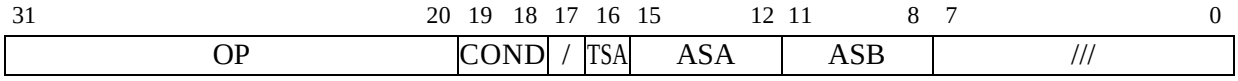
- NEQ is 1 and address registers ASA is zero, or
- NEQ is 0 and address registers ASA is not zero

set the value of the program counter to the contents of the branch target register TSA.

In the assembler, the mnemonics `jcaz_ne`/ `jcaz_eq` are used for NEQ=0/1

2.3.4.3 Jump Address Conditional

`jcau_cond $tsa,$asa,$asb`




```
if( ((COND = 00b) && [ASA] >u [ASB])
||  ((COND = 01b) && [ASA] <u [ASB])
||  ((COND = 10b) && [ASA] ≤u [ASB])
||  ((COND = 11b) && [ASA] ≥u [ASB]) )
  PC ← [TSA]
else
  PC ← PC + 4
```

The contents of address registers ASA and ASB are compared to each other as unsigned values. The COND specifies the test.

- COND = 0: unsigned greater than
- COND = 1: unsigned less than
- COND = 2: unsigned less than or equal to
- COND = 3: unsigned greater than or equal to

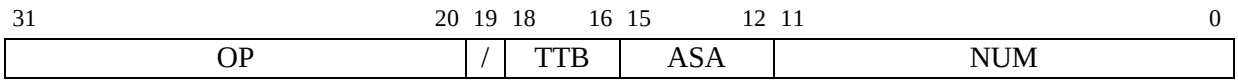
If the test is satisfied, set the value of the program to the contents of the branch target register TSA.

In the assembler, the mnemonics `jcau_gt/ jcau_lt/ jcau_le/ jcau_ge` are used for COND=0/1/2/3

2.3.5 Load/Store Instructions

2.3.5.1 Load Branch Target Word

`lditw $ttb,$asa,num`



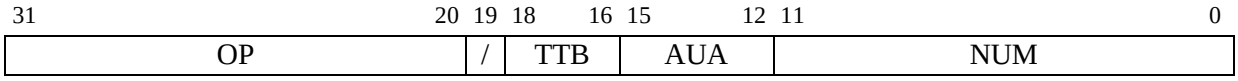

```
TTB ← MEM( [ASA] + NUM )
```

Load the target register with the 32-bit value stored in memory at the address in ASA incremented by sign-extended value of the NUM.

The results of this operation are undefined if the address is not 4B aligned, or if the value stored into TTB is not 4B aligned. The results of this operation are undefined if TTB is not one of 0, 1, 4 or 7.

2.3.5.2 Load Branch Target Word and Update

`ldutw $ttb,$aua,num`



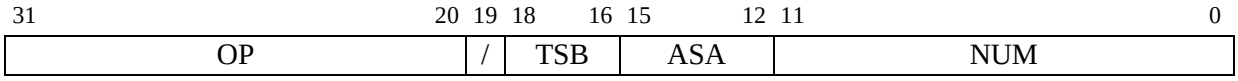

```
TTB ← MEM( [AUA] )
AUA ← [AUA] + NUM
```

Load the target register with the 32-bit value stored in memory at the address in AUA. Increment AUA by the sign-extended value of the NUM.

The results of this operation are undefined if the address is not 4B aligned, or if the value stored into TTB is not 4B aligned. The results of this operation are undefined if TTB is not one of 0, 1, 4 or 7.

2.3.5.3 Store Branch Target Word

stitw $tsb,$asa,num



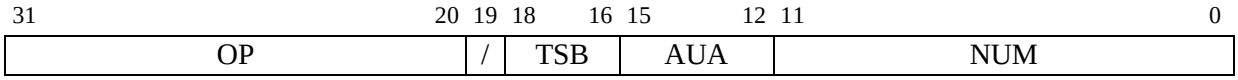
MEM([ASA] + NUM) ← [TSB]

Store the 32-bit value stored in the target register to memory at the address in ASA incremented by sign-extended value of the NUM.

The results of this operation are undefined if the address is not 4B aligned. The results of this operation are undefined if TSB is not one of 0, 1, 4 or 7.

2.3.5.4 Store Branch Target Word and Update

stutw $tsb,$aua,num



MEM([AUA]) ← [TSB]
AUA ← [AUA] + NUM

Store the 32-bit value stored in the target register in memory to the address in ASA. Increment AUA by the sign-extended value of the NUM.

The results of this operation are undefined if the address is not 4B aligned. The results of this operation are undefined if TSB is not one of 0, 1 4 or 7.

3 Address/Load-Store Unit

3.1 Overview

In the Unity architecture, the address for all data memory operations shall be provided by the address registers, `$a0…$a15`. All load/store operations are memory-to-register/register-to-memory copy operations; the only operations performed on the value as it is moving to/from memory are operations such as sign-extension and other reformatting of values.

Load/Stores also support base plus offset address, where a 12 bit signed offset is added to the address register to generate final memory address.

In this chapter, we shall describe the instructions that operate on the address registers. Load/stores to non-address registers are described in the sections dealing with those registers. Conditional jumps based on the address registers are described in the chapter on the branch unit.

3.2 Memory operations

All memory accesses must be aligned to the size of the data being accessed[4]. Thus, a load of a 32-bit register is expected to reference a 4B aligned address (i.e. one with the lowest 2 bits 0).

Memory access is little-endian order.

3.3 Address Registers

The address unit has 16 32-bit address registers, written `$a0…$a15`. Of these registers, 11 (`$a0…$a10`) are user accessible and 5 (`$a11…$a15`) can be accessed only at hypervisor level.

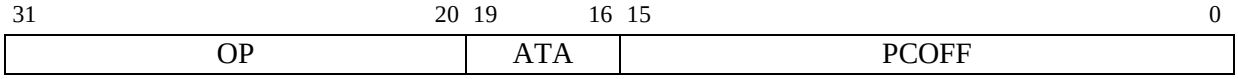
[4] Vector loads and stores are a special case.

3.4 Address and Load/Store Instructions

3.4.1 Address Arithmetic Instructions

3.4.1.1 Set Address to PC offset

aaddpci $ata,num



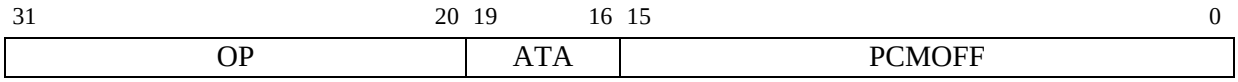
ATA ← PC + PCOFF<<2

The address register ATA is set to the sum of the program counter of the instruction and the sign-extended value of the 16 bit immediate PCOFF shifted by 2.

This instruction is used to allow for PC relative storage of variables.

3.4.1.2 Set Address to PC offset Medium

aaddpcimb $ata,pcmoff

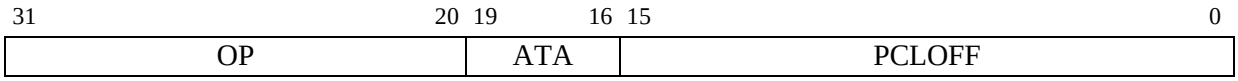


ATA ← $(PC_{31:12}$ + PCOFF<<12$).0_{11:0}$

The address register ATA is set to the sum of the program counter of the instruction and the sign-extended value of the 16 bit immediate PCMOFF shifted by 12. The lower 12 bits are zeroed

3.4.1.3 Set Address to PC offset Long

aaddpcilb $ata,pcloff

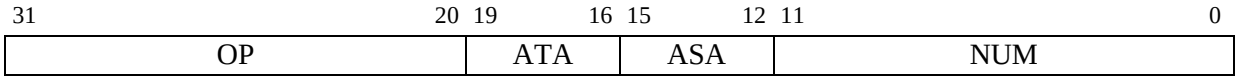


ATA ← $(PC_{31:15}$ + PCOFF<<15$).0_{14:0}$

The address register ATA is set to the sum of the program counter of the instruction and the sign-extended value of the 16 bit immediate PCLOFF shifted by 15. The lower 15 bits are zeroed

3.4.1.4 Add Address Immediate

aaddai $ata,$asa,num

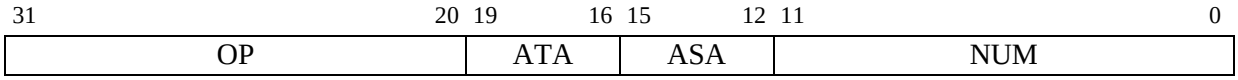


ATA ←[ASA] + NUM

The address register ATA is set to the sum of the contents of address register ASA and the sign-extended immediate field NUM.

3.4.1.5 AND Address Immediate

aandai $ata,$asa,num

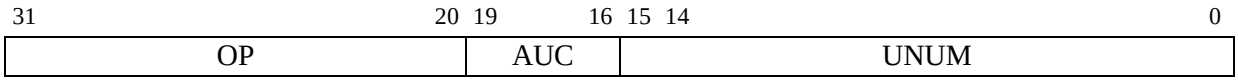


ATA ←[ASA] & NUM

The address register ATA is set to the bitwise AND of the contents of address register ASA and the sign-extended immediate field NUM.

3.4.1.6 Add Address Update Immediate

aadduib $auc,unum

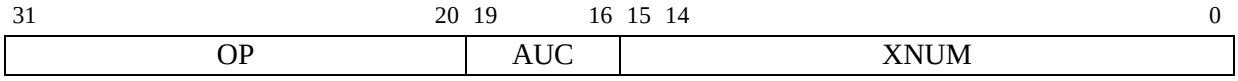


AUC ←[AUC] + UNUM

The address register AUC is set to the sum of the contents of its previous contents and the sign-extended 16-bit immediate field UNUM.

3.4.1.7 Add Address Update Immediate Extended

aadduix $auc,xnum

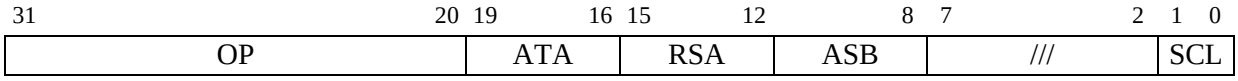



```
AUC ←[AUC] + (XNUM<<12)
```

The address register AUC is set to the sum of the contents of its previous contents and the sign-extended 16-bit immediate field XNUM shifted by 12 bits.

3.4.1.8 Set Address from Address and Register

aaddar_scl $ata,$asb,$rsa



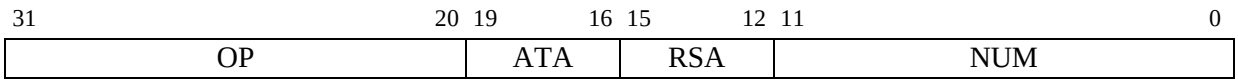

```
ATA ←[ASB] + [RSA]<<SCL
```

The address register ATA is set to the sum of the contents of address register ASB and general-purpose register RSA shifted by 0/1/2 or 3 bits, based on the SCL field.

In the assembler, the mnemonics `aaddar_b`/ `aaddar_h`/ `aaddar_w`/ `aaddar_l` are used for SCL=0/1/2/3

3.4.1.9 Set Address from Register Offset

aaddri $ata,$rsa,num

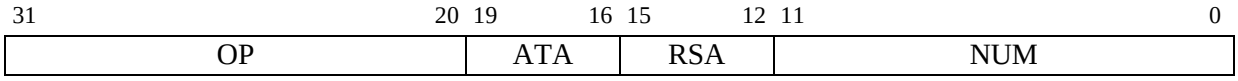



```
ATA ←[RSA] + NUM
```

The address register ATA is set to the sum of the contents of general-purpose register RSA and the sign-extended 12-bit immediate field NUM.

3.4.1.10 Set Address from AND Register Immediate

aandri $ata,$rsa,num

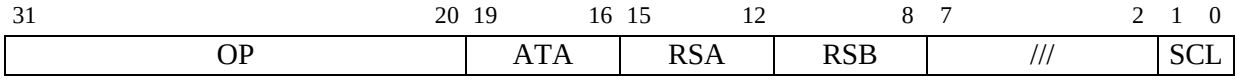



```
ATA ←[RSA] & NUM
```

The address register ATA is set to the bitwise AND of the contents of general-purpose register RSA and the sign-extended 12-bit immediate field NUM

3.4.1.11 Set Address from Registers

aaddrr_scl $ata,$rsa,$rsb



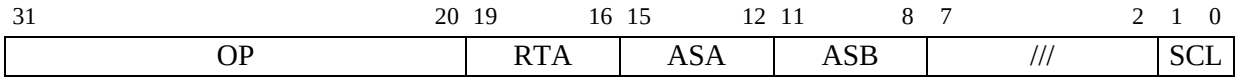

```
ATA ←[RSB] + [RSA]<<SCL
```

The address register ATA is set to the sum of the contents of address register RSB and general-purpose register RSA shifted by 0/1/2 or 3 bits, based on the SCL field.

In the assembler, the mnemonics `aaddrr_b`/ `aaddrr_h`/ `aaddrr_w`/ `aaddrr_l` are used for SCL=0/1/2/3

3.4.1.12 Set GPR from Address Subtract

rsubaa_scl $rta,$asa,$asb



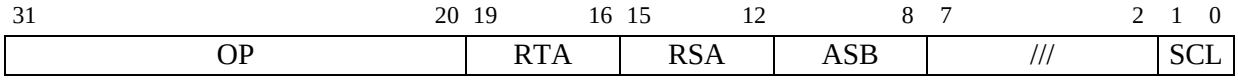

```
RTA ← ( [ASA] - [ASB] )>>SCL
```

The general-purpose register RTA is set to the result of subtracting the contents of address register ASB from address register ASA, right shifted (with sign extension) by 0/1/2/3 depending on SCL.

In the assembler, the mnemonics `rsubaa_b`/ `rsubaa_h`/ `rsubaa_w`/ `rsubaa_l` are used for SCL=0/1/2/3

3.4.1.13 Set GPR from Address Register

raddar_scl $rta,$rsa,$asb



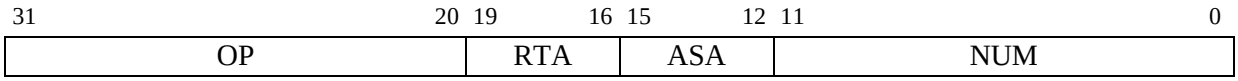
RTA ←[ASB] + [RSA]<<SCL

The general-purpose register RTA is set to the sum of the contents of address register ASB and general-purpose register RSA shifted by 0/1/2 or 3 bits, based on the SCL field.

In the assembler, the mnemonics `raddar_b`/ `raddar_h`/ `raddar_w`/ `raddar_l` are used for SCL=0/1/2/3

3.4.1.14 Set GPR from Address Offset

raddai $rta,$asa,num



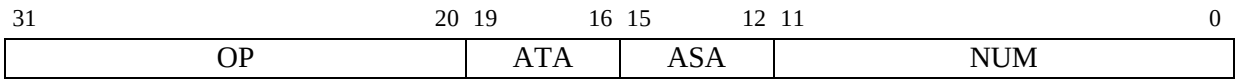
RTA ← [ASA] + NUM

The general-purpose register RTA is set to the sum of the contents of address register ASA and the sign-extended 12-bit immediate field NUM.

3.4.2 Load/Store Instructions

3.4.2.1 Load Address

`ldia $ata,$asa,num`



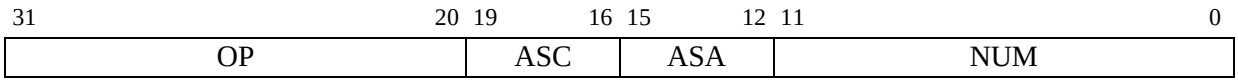

```
[ATA + 1].[ATA] ← MEM( [ASA] + NUM )
```

Load the address register pair ATA with the 64-bit value stored in memory at the address in ASA incremented by sign-extended value of immediate NUM. ATA must be an even register address.

The results of this operation are undefined if the address is not 8B aligned.

3.4.2.2 Store Address

`stia $asc,$asa,num`



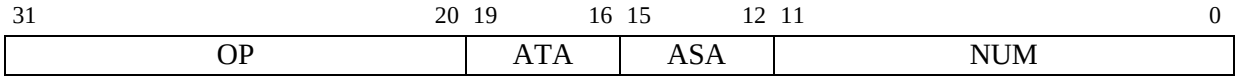

```
MEM( [ASA] + NUM ) ← [ASC+1].[ASC]
```

Store the 64-bit value stored in the address register pair ASC to the memory at the address in ASA incremented by sign-extended value of immediate NUM. ASC must be an even register address.

The results of this operation are undefined if the address is not 8B aligned.

3.4.2.3 Load Address Word

ldiaw $ata,$asa,num



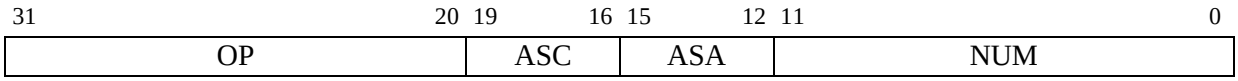
ATA ← MEM([ASA] + NUM)

Load the address register ATA with the 32-bit value stored in memory at the address in ASA incremented by sign-extended value of immediate NUM.

The results of this operation are undefined if the address is not 4B aligned.

3.4.2.4 Store Address Word

stiaw $asc,$asa,num



MEM([ASA] + NUM) ← [ASC]

Store the 32-bit value stored in the address register ASC to the memory at the address in ASA incremented by sign-extended value of immediate NUM.

The results of this operation are undefined if the address is not 4B aligned.

4 Integer Unit

4.1 Overview

This chapter describes the general-purpose registers used in the integer unit and most of the instructions that access them. Instructions that interact with other register groups, such as conditional jumps, and operations that copy to/from the general-purpose registers are described in other chapters.

4.2 Integer Registers

4.2.1 General-purpose Registers

There are 16 32-bit general-purpose registers, generally written as `$r0`...`$r15`.

4.2.2 Fixcr Register

The 2 bits of the rounding mode control are interpreted as:

- 00: Round to nearest, half to even
- 01: Round to -∞
- 10: Round to +∞
- 11: Round to 0

4.2.3 Carry/Saturate/Condition flag

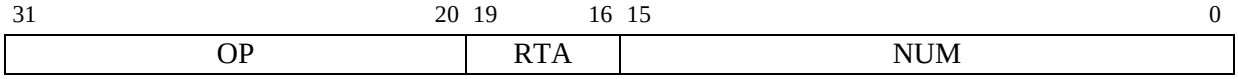
There is a single 1-bit flag available, called the carry/saturate/condition flag, that is accessed by carry/borrow instructions, saturating instructions, comparison instructions and conditional selection instructions.

4.3 Integer Instructions

4.3.1 Immediate Instructions

4.3.1.1 Set GPR to immediate

rseti $rta,num

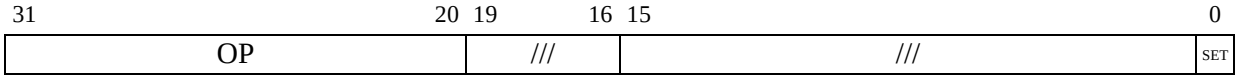


RTA ← NUM

The general-purpose register RTA is set to the sign-extended value of the 16 immediate NUM.

4.3.1.2 Set CF to immediate

cfseti_set

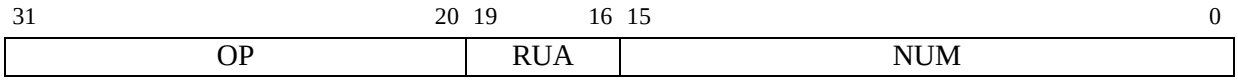


CF ← SET

Set cf register to 0 or 1.

4.3.1.3 Insert and update immediate

insui $rua,num



RUA ← $RUA_{15:0}$. NUM

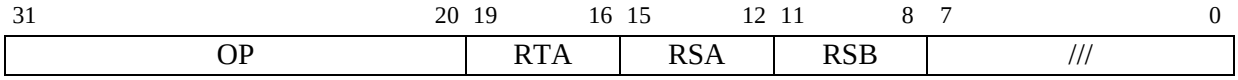
The general-purpose register RUA is set to concatenation of its lower 16 bits and the 16-bit immediate NUM.

`rseti` followed by an `inusi` instructions can be used to create signed values of length 32 bits.

4.3.2 Add/Subtract Instructions

4.3.2.1 Add

`add $rta,$rsa,$rsb`

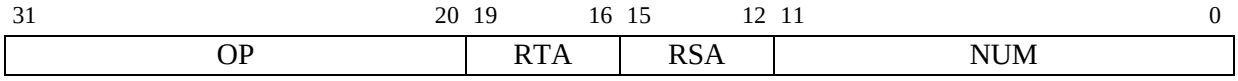


`RTA ← [RSA] + [RSB]`

The general-purpose register RTA is set to the sum of the contents of general-purpose registers RSA and RSB.

4.3.2.2 Add Immediate

`addi $rta,$rsa,num`

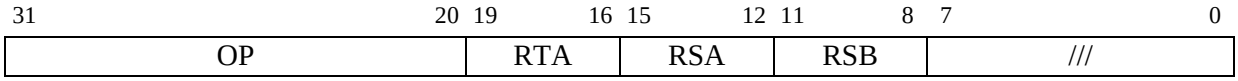


`RTA ← [RSA] + NUM`

The general-purpose register RTA is set to the sum of the contents of general-purpose registers RSA and the sign-extended value of the immediate NUM.

4.3.2.3 Add with carry

`addc $rta,$rsa,$rsb`

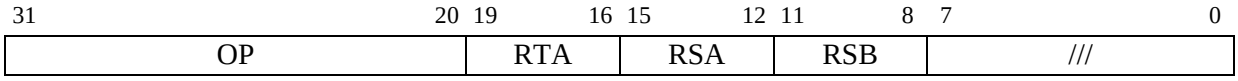


`CA,RTA ← [RSA] + [RSB] + CA`

The general-purpose register RTA is set to the sum of the contents of general-purpose registers RSA and RSB and the carry bit. The carry bit CA is set to 1 if this sum generates a carry, otherwise 0.

4.3.2.4 Subtract from

subf $rta,$rsa,$rsb

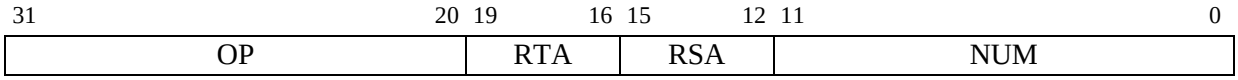


RTA ← [RSB] - [RSA]

The general-purpose register RTA is set to result of subtracting the contents of general-purpose register RSA from the contents of RSB.

4.3.2.5 Subtract from Immediate

subfi $rta,$rsa,num

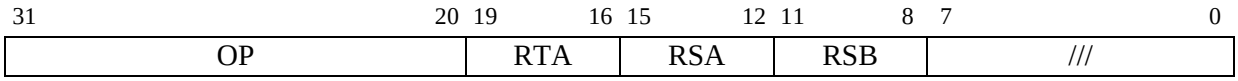


RTA ← NUM - [RSA]

The general-purpose register RTA is set to the result of subtracting the contents of general-purpose register RSA from the sign-extended value of the immediate NUM.

4.3.2.6 Subtract from with borrow

subfb $rta,$rsa,$rsb

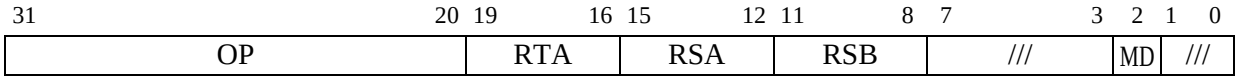


CA,RTA ← [RSB] - [RSA] - CA

The general-purpose register RTA is set to the result of subtracting the contents of general-purpose register RSA and the carry bit from the contents of RSB. The carry bit is set to 1 if this sum generates a borrow, otherwise 0.

4.3.2.7 Maximum

max_md $rta,$rsa,$rsb



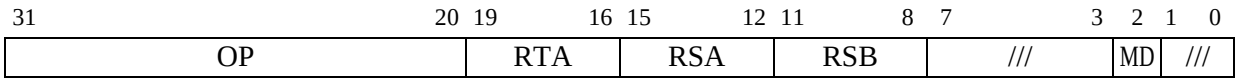

```
If( MD = 0 )
  RTA ← if( [RSA] > [RSB] ) [RSA] else [RSB]
else
  RTA ← if( [RSA] >unsigned [RSB] ) [RSA] else [RSB]
```

The general-purpose register RTA is set to the maximum of the contents of general-purpose registers RSA and RSB. Both numbers treated as signed numbers if MD = 0 otherwise they are treated as unsigned numbers.

In the assembler, the mnemonics `max_u`/ `max_s` are used for MD=1/0. `max` is equivalent to `max_s`

4.3.2.8 Minimum

min_md $rta,$rsa,$rsb




```
If( MD = 0 )
  RTA ← if( [RSA] < [RSB] ) [RSA] else [RSB]
else
  RTA ← if( [RSA] <unsigned [RSB] ) [RSA] else [RSB]
```

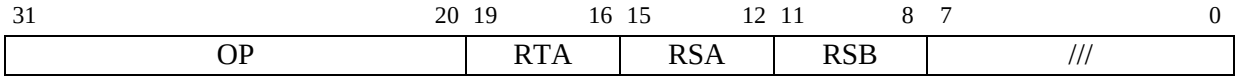
The general-purpose register RTA is set to the minimum of the contents of general-purpose registers RSA and RSB. Both numbers treated as signed numbers if MD = 0 otherwise they are treated as unsigned numbers

In the assembler, the mnemonics `min_u`/ `min_s` are used for MD=1/0. `min` is equivalent to `min_s`

4.3.3 Logical Instructions

4.3.3.1 And

`and $rta,$rsa,$rsb`

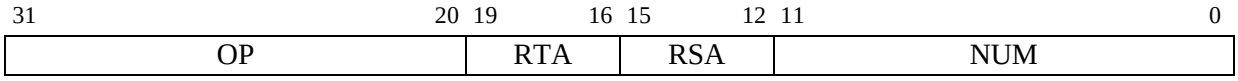


`RTA ← [RSA] & [RSB]`

The general-purpose register RTA is set to the bitwise AND of the contents of general-purpose registers RSA and RSB.

4.3.3.2 And Immediate

`andi $rta,$rsa,num`

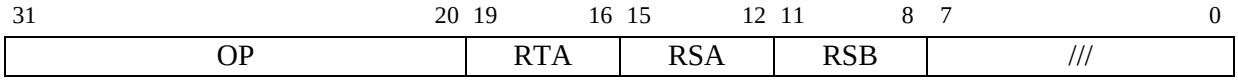


`RTA ← [RSA] & NUM`

The general-purpose register RTA is set to the bitwise AND of the contents of general-purpose register RSA and the sign-extended value of the immediate NUM.

4.3.3.3 Cand

`cand $rta,$rsa,$rsb`

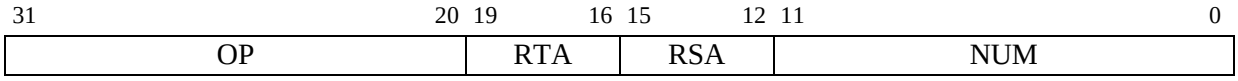


`RTA ← ~([RSA]) & [RSB]`

The general-purpose register RTA is set to the bitwise AND of the contents of general-purpose registers RSA complemented with the contents of RSB.

4.3.3.4 Cand Immediate

candi $rta,$rsa,num

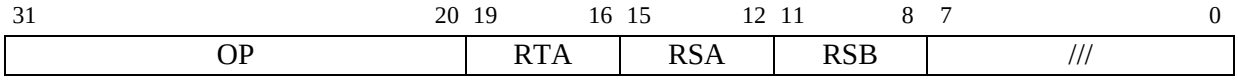



```
RTA ← ~([RSA]) & NUM
```

The general-purpose register RTA is set to the bitwise AND of the contents of general-purpose-register RSA complemented and the sign-extended value of the immediate NUM.

4.3.3.5 Or

or $rta,$rsa,$rsb

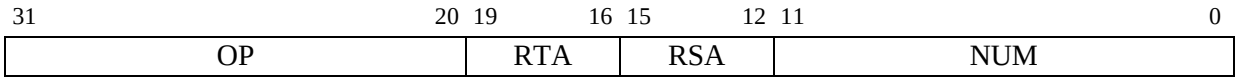



```
RTA ← [RSA] | [RSB]
```

The general-purpose register RTA is set to the bitwise OR of the contents of general-purpose registers RSA and RSB.

4.3.3.6 Or Immediate

ori $rta,$rsa,num

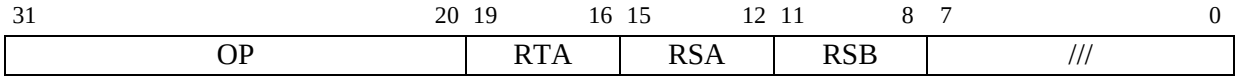



```
RTA ← [RSA] | NUM
```

The general-purpose register RTA is set to the bitwise or of the contents of general-purpose register RSA and the sign-extended value of the immediate NUM.

4.3.3.7 Xor

xor $rta,$rsa,$rsb

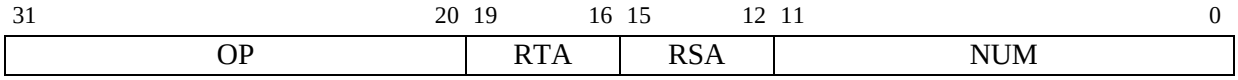


RTA ← [RSA] ^ [RSB]

The general-purpose register RTA is set to the bitwise exclusive OR of the contents of general-purpose registers RSA and RSB.

4.3.3.8 Xor Immediate

xori $rta,$rsa,num



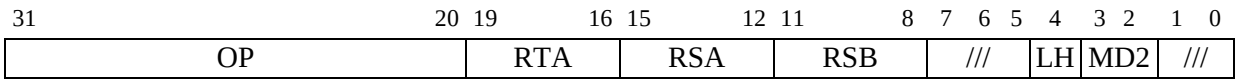
RTA ← [RSA] ^ NUM

The general-purpose register RTA is set to the bitwise exclusive OR of the contents of general-purpose registers RSA and the sign-extended value of the immediate NUM.

4.3.4 Multiply and Divide Instructions

4.3.4.1 Multiply

mul_lh_md2 $rta,$rsa,$rsb



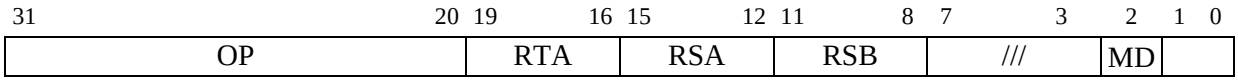

```
va ← [RSA]
vb ← [RSB]

If( MD2 = 00b )
  vt ← signed(va) * signed(vb)
else if( MD2 = 01b )
  vt ← signed(va) * unsigned(vb)
else if( MD2 = 10b )
  vt ← unsigned(va) * signed(vb)
else if( MD2 = 11b )
  vt ← unsigned(va) * unsigned(vb)

If(LH=0b)
  RTA ← vt31:0
 else
  RTA ← vt63:32
```

The general- purpose register RTA is set to half of the 64-bit result of doing a multiply of RSA and RSB, depending on the value of the LH field, either the upper half or lower half 32 bits are used. The two operands can be treated as signed or unsigned, based on the value of the MD2 field.

In the assembler, the mnemonics `mul_lh_uu`/ `mul_lh_us` /`mul_lh_su` /`mul_lh_ss` are used for MD2=00/01/10/11, the mnemonics `mul_l_md2`/ `mul_h_md2` are used for LH=0/1.

4.3.4.2 Divide

div_md $rta,$rsa,$rsb




```
If( MD = 0b )
  RTA ← signed([RSA]) / signed([RSB])
else if( MD = 1b )
  RTA ← unsigned([RSA]) / unsigned([RSB])
```

The general- purpose register RTA is set to the quotient of dividing the contents of RSA by the contents of RSB. If MD=0, both numbers treated as signed numbers, otherwise as unsigned.

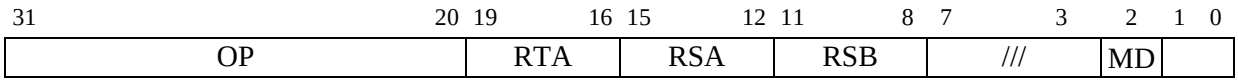
An integer divide exception is raised if the following are true:

- divisor is 0
- MD=0 (signed division), dividend is most negative number and divisor is -1

In the assembler, the mnemonics `div_u`/ `div_s` are used for MD=1/0.

4.3.4.3 Remainder

`rem_md $rta,$rsa,$rsb`




```
If( MD = 0b )
  RTA ← signed([RSA]) % signed([RSB])
else if( MD = 1b )
  RTA ← unsigned([RSA]) % unsigned([RSB])
```

The general- purpose register RTA is set to the remainder of dividing the contents of RSA by the contents of RSB. If MD=0, both numbers treated as signed numbers, otherwise as unsigned. For signed division, the sign of the remainder is the same as the sign of the divisor

An integer divide exception is raised if the following are true:

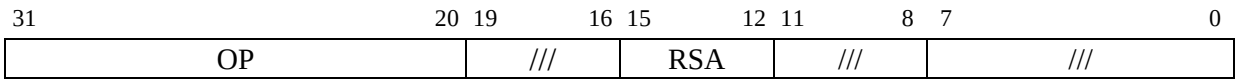
- divisor is 0
- MD=0 (signed division), dividend is most negative number and divisor is -1

In the assembler, the mnemonics `rem_u`/ `rem_s` are used for MD=1/0

4.3.5 Fix Point Instructions

4.3.5.1 Fix control register set from GPR

fcrsetr $rsa

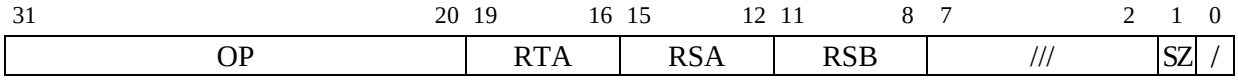


FIXCR ← [RSA]$_{1:0}$

The fix point control register FIXCR is set to the lowest two bit of RSA.

4.3.5.2 Fix Point Multiply

Fixmul_sz $rta,$rsa,$rsb



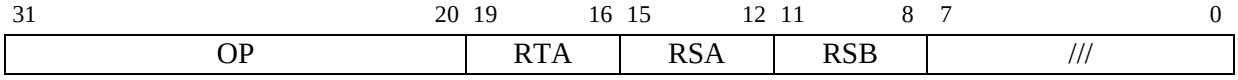

```
If(sz = 0)
  temp ← fix-point1/15[RSA]15:0 * fix-point1/15[RSB]15:0
  if( temp = 1 )
    RTA ← 0x00007FFF
    CA ← 1
  else
    RTA ← temp
Else
  temp ← fix-point1/31[RSA] * fix-point1/31[RSB]
  if( temp = 1 )
    RTA ← 0x7FFFFFFF
    CA ← 1
  else
    RTA ← temp
```

Multiply contents of RSA and RSB as fixed-point numbers of the lower 16-bit or 32-bit depend on SZ field, round based on the control specified in FIXCR and store the result in RTA.

In the assembler, the mnemonics `fixmul_w`/ `fixmul_h` are used for SZ=1/0

4.3.5.3 Fix point Divide

fixdiv $rta,$rsa,$rsb



```
temp ← fix-point1/15[RSA]15:0 / fix-point1/15[RSB]15:0
if( temp >= 1 )
  RTA ← 0x00007FFF
  CA ← 1
else
  RTA ← temp
```

Dividing the 16-bit fixed-pointer numbers of RSA by the contents of RSB, round based on the control specified in FIXCR and store the quotient in RTA.

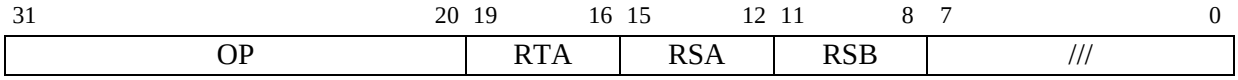
An integer divide exception is raised if the following are true:

- divisor is 0

4.3.6 Shift Word Instructions

4.3.6.1 Shift left word

shlw $rta,$rsa,$rsb

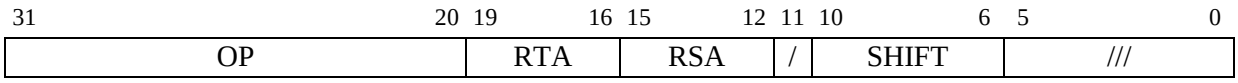


RTA ← [RSA] << $[RSB]_{4:0}$

The general-purpose register RTA is set to the result of left shifting the 32 bits of the contents of general-purpose register RSA by the amount specified by the lowest 5 bits of RSB.

4.3.6.2 Shift left word immediate

shlwi $rta,$rsa,shift

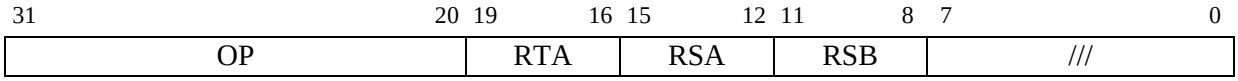


RTA ← [RSA] << SHIFT

The general-purpose register RTA is set to the result of left shifting the contents of general-purpose register RSA by the amount specified by the 5 bit immediate SHIFT.

4.3.6.3 Shift right word

shrw $rta,$rsa,$rsb

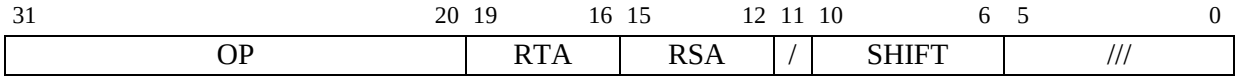


RTA ← [RSA] >> $[RSB]_{4:0}$

The general-purpose register RTA is set to the result of right shifting the 32 bits of the contents of general-purpose register RSA by the amount specified by the lowest 5 bits of RSB. 0s are shifted in to the high bits of the result.

4.3.6.4 Shift right word immediate

shrwi $rta,$rsa,shift

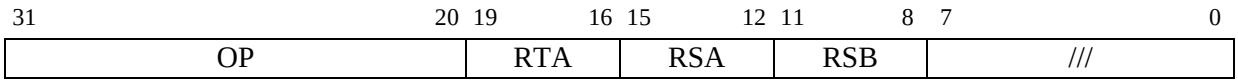


RTA ← [RSA] >> SHIFT

The general-purpose register RTA is set to the result of right shifting the 32 bits of the contents of general-purpose register RSA by the amount specified by the 5 bits of the immediate SHIFT. 0s are shifted in to the high bits of the result.

4.3.6.5 Shift right arithmetic word

shraw $rta,$rsa,$rsb

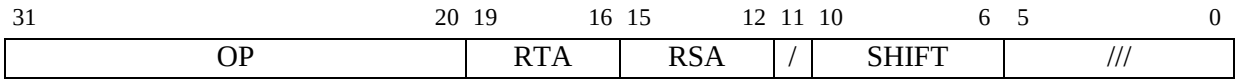


RTA ← [RSA] >>$_{arith}$ [RSB]$_{4:0}$

The general-purpose register RTA is set to the result of right shifting the 32 bits of the contents of general-purpose register RSA by the amount specified by the lowest 5 bits of RSB. The sign bit of RSA is shifted in to the high bits of the result.

4.3.6.6 Shift right arithmetic word immediate

shrawi $rta,$rsa,shift



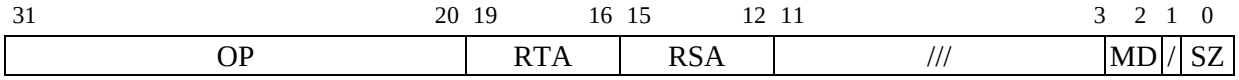
RTA ← [RSA] >>$_{arith}$ SHIFT

The general-purpose register RTA is set to the result of right shifting the contents of general-purpose register RSA by the amount specified by the 5 bit immediate SHIFT. The sign bit of RSA is shifted in to the high bits of the result.

4.3.7 Sign Extend Instructions

4.3.7.1 Extend signed

ext_md_sz $rta,$rsa




```
if( MD = 0 )
  if( SZ = 1 )
    RTA ← signed( [RSA]15:0 )
  else if( SZ = 0 )
    RTA ← signed( [RSA]7:0 )
else
  if( SZ = 1 )
    RTA ← [RSA]15:0
  else if( SZ = 0 )
    RTA ← [RSA]7:0
```

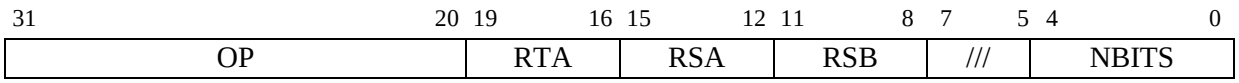
The general-purpose register RTA is set to the result of extending the lower 16/8 bits of the contents of RSA, based on the value of the SZ field. The extension is sign-extended or zero-extended, based on the value of MD.

In the assembler, the mnemonics `ext_md_h`/ `ext_md_b` are used for SZ=1/0. The mnemonics `ext_u_sz`/ `ext_s_sz` are used for MD=1/0.

4.3.8 Extract Instructions

4.3.8.1 Extract unsigned

extru $rta,$rsa,$rsb,nbits



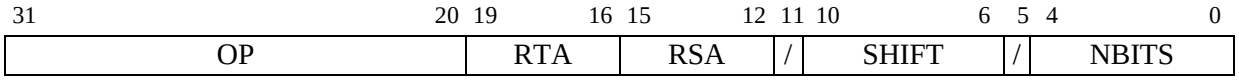
RTA ← ([RSA] >> [RSB]$_{4:0}$)$_{NBITS-1:0}$

The general-purpose register RTA is set to the result of right shifting the contents of general-purpose register RSA by the amount specified by the lowest 5 bits the general-purpose register RSB, and then extracting the lowest NBITS of the result. 0s are shifted in to the high bits of the result.

The behavior when NBITS is 0 is undefined. The value when the shift + NBITS is greater than 32 is undefined.

4.3.8.2 Extract unsigned immediate

extrui $rta,$rsa,shift,nbits



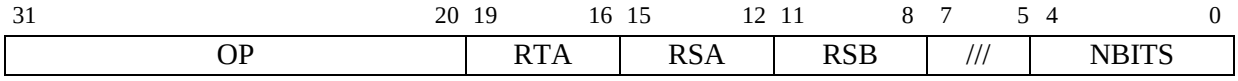
RTA ← ([RSA] >> SHIFT)$_{NBITS-1:0}$

The general-purpose register RTA is set to the result of right shifting the contents of general-purpose register RSA by the amount specified by the 5 bits of the immediate SHIFT, and then extracting the lowest NBITS of the result. 0s are shifted in to the high bits of the result

The behavior when NBITS is 0 is undefined. The value when the shift + NBITS is greater than 32 is undefined.

4.3.8.3 Extract signed

extrs $rta,$rsa,$rsb,nbits



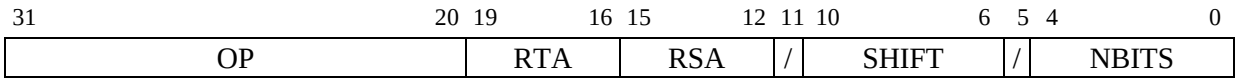
RTA ← signextend(([RSA] >> [RSB]$_{4:0}$)$_{NBITS-1:0}$)

The general-purpose register RTA is set to the result of right shifting the contents of general-purpose register RSA by the amount specified by the lowest 5 bits the general-purpose register RSB, and then extracting the lowest NBITS of the result and sign-extending it to 32 bits by copying the NBITS-1 bit.

The behavior when NBITS is 0 is undefined. The value when the shift + NBITS is greater than 32 is undefined.

4.3.8.4 Extract signed immediate

extrsi $rta,$rsa,shift,nbits



RTA ← signextend(([RSA] >> SHIFT)$_{NBITS-1:0}$)

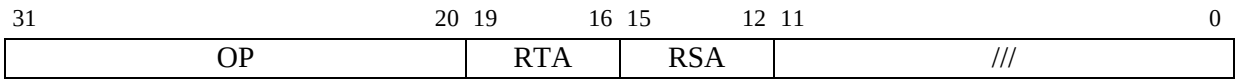
The general-purpose register RTA is set to the result of right shifting the contents of general-purpose register RSA by the amount specified by the 5 bits of the immediate SHIFT, and then extracting the lowest NBITs of the result and sign-extending it to 32 bits by copying the NBITS-1 bit.

The behavior when NBITS is 0 is undefined. The value when the shift + NBITS is greater than 32 is undefined.

4.3.9 Bit/Byte Instructions

4.3.9.1 Swap byte

swap $rta,$rsa

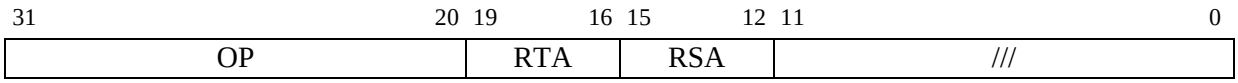



```
temp ← [RSA]
RTA ← temp7:0 . temp15:8 . temp23:16 . temp31:24
```

The general-purpose register RTA is set to the value of general-purpose register RSA with bytes in reverse order.

4.3.9.2 Reverse bits

rev $rta,$rsa



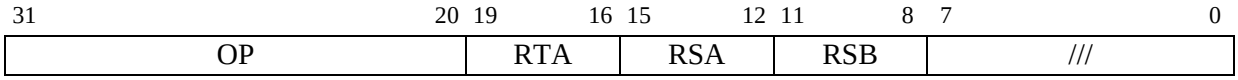

```
temp ← [RSA]
for( i in 0:31 )
  resi ← temp31-i
RTA ← res
```

The general-purpose register RTA is set to the value of general-purpose register RSA with bits in reverse order

4.3.10 Count Instructions

4.3.10.1 Count non-zero bytes

countnzb $rta,$rsa,$rsb

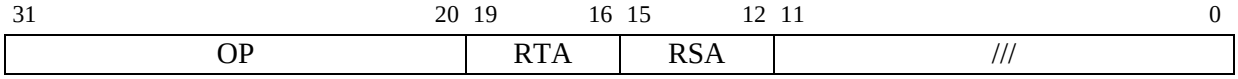



```
tempA ← [RSA]
tempB ← [RSB]
res ← 0
for( i in 0:3 )
  if( tempA8*i+7:8*i ≠ 0 && tempB8*i+7:8*i ≠ 0)
    res ← res + 1
RTA ← res
```

The general-purpose register RTA is set to the number of non-zero bytes in the value produced by bit-wise and-ing general-purpose registers RSA and RSB.

4.3.10.2 Count set bits

countsb $rta,$rsa

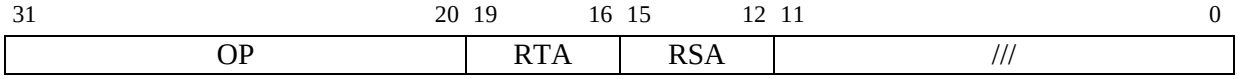



```
temp ← [RSA]
res ← 0
for( i in 0:31 )
  res ← res + tempi
RTA ← res
```

The general-purpose register RTA is set to the total number of 1 bits in the value of general-purpose register RSA.

4.3.10.3 Count leading zero bits

countlz $rta,$rsa

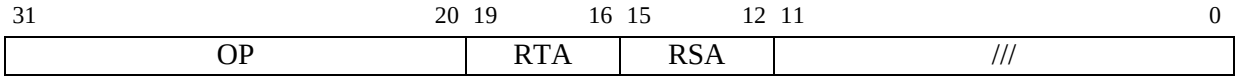


```
temp ← [RSA]
res ← 0
for( i in 31:0 )
  if( tempi ≠ 0b )
    break;
  res ← res + 1
RTA ← res
```

The general-purpose register RTA is set to the number of leading 0 bits in the value of general-purpose register RSA.

4.3.10.4 Count leading sign bits

countls $rta,$rsa

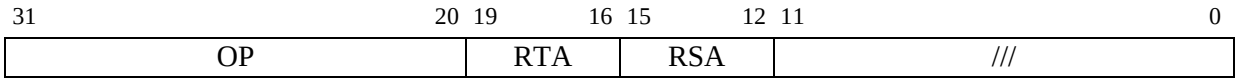



```
temp ← [RSA]
sign ← temp31
res ← 1
for( i in 30:0 )
  if( tempi ≠ sign )
    break;
  res ← res + 1
RTA ← res
```

The general-purpose register RTA is set to the number of leading bits (after and not including the sign bit) in the value of general-purpose register RSA that are equal to its sign bit.

4.3.10.5 Count trailing zero bits

counttz $rta,$rsa



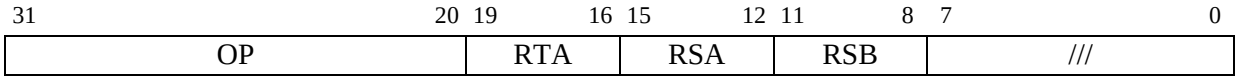

```
temp ← [RSA]
res ← 0
for( i in 0:31 )
  if( tempi ≠ 0b )
    break;
  res ← res + 1
RTA ← res
```

The general-purpose register RTA is set to the number of trailing 0 bits in the value of general-purpose register RSA.

4.3.11 Saturating Instructions

4.3.11.1 Add saturating

adds $rta,$rsa,$rsb

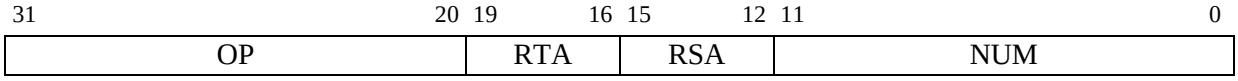



```
temp ← [RSA] + [RSB]
if( temp < -2147483648 )
  CA ← 1
  RTA ← -2147483648
else if( temp > 2147483647 )
  CA ← 1
  RTA ← 2147483647
else
  CA ← 0
  RTA ← temp
```

The general-purpose register RTA is set to the sum of the 32 bits of general-purpose registers RSA and RSB, saturated to 32 bits.

4.3.11.2 Add saturating immediate

addsi $rta,$rsa,$rsb

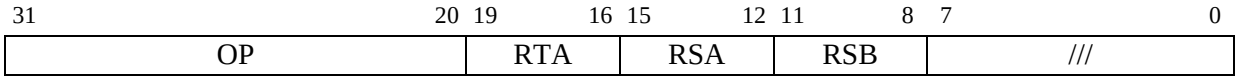



```
temp ← [RSA] + NUM
if( temp < -2147483648 )
  CA ← 1
  RTA ← -2147483648
else if( temp > 2147483647 )
  CA ← 1
  RTA ← 2147483647
else
  CA ← 0
  RTA ← temp
```

The general-purpose register RTA is set to the sum of the 32 bits of general-purpose registers RSA with the sign-extended immediate NUM, saturated to 32 bits.

4.3.11.3 Subtract from saturating

subfs $rta,$rsa,$rsb

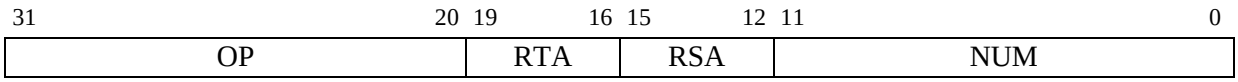



```
temp ← [RSB] - [RSA]
if( temp < -2147483648 )
  CA ← 1
  RTA ← -2147483648
else if( temp > 2147483647 )
  CA ← 1
  RTA ← 2147483647
else
  CA ← 0
  RTA ← temp
```

The general-purpose register RTA is set to the value of the 32 bits of general-purpose registers RSA subtracted from the 32 bits of RSB, saturated to 32 bits.

4.3.11.4 Subtract from saturating immediate

subfsi $rta,$rsa,num

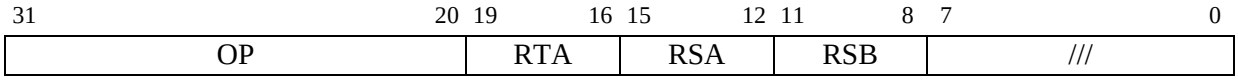



```
temp ← NUM - [RSA]
if( temp < -2147483648 )
  CA ← 1
  RTA ← -2147483648
else if( temp > 2147483647 )
  CA ← 1
  RTA ← 2147483647
else
  CA ← 0
  RTA ← temp
```

The general-purpose register RTA is set to the value of the 32 bits of general-purpose registers RSA subtracted from the sign-extended immediate NUM, saturated to 32 bits.

4.3.11.5 Shift left saturating

shls $rta,$rsa,$rsb

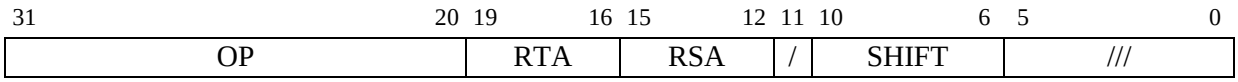



```
temp ← [RSA] << [RSB]4:0
if( temp < -2147483648 )
  CA ← 1
  RTA ← -2147483648
else if( temp > 2147483647 )
  CA ← 1
  RTA ← 2147483647
else
  CA ← 0
  RTA ← temp
```

The general-purpose register RTA is set to the value of the 32 bits of general-purpose registers RSA shifted by the lower 5 bits of RSB, saturated to 32 bits.

4.3.11.6 Shift left saturating immediate

shlsi $rta,$rsa,shift

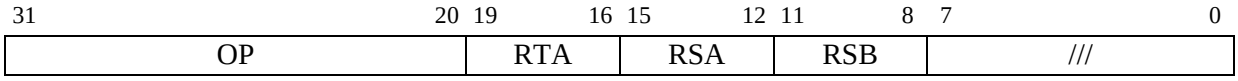



```
temp ← [RSA] << SHIFT
if( temp < -2147483648 )
  CA ← 1
  RTA ← -2147483648
else if( temp > 2147483647 )
  CA ← 1
  RTA ← 2147483647
else
  CA ← 0
  RTA ← temp
```

The general-purpose register RTA is set to the value of the 32 bits of general-purpose register RSA shifted by SHIFT, saturated to 32 bits.

4.3.11.7 Multiply saturating

mulsat $rta,$rsa,$rsb

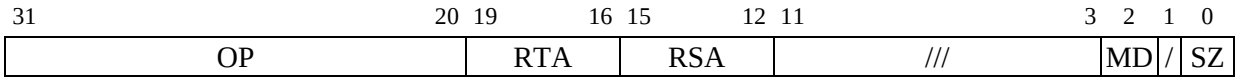



```
temp ← signed([RSA]15:0)* signed([RSB]15:0)*2
if( temp > 2147483647 )
  CA ← 1
  RTA ← 2147483647
else
  CA ← 0
  RTA ← temp
```

The general-purpose register RTA is set to the 32-bit result of doing a multiply of the lower 16 bits of RSA and RSB, both treated as signed numbers, times 2, saturated to 32 bits.

4.3.11.8 Saturate word/half-word/byte

sat_md_sz $rta,$rsa




```
temp ← RSA
if( MD = 0 )
  if( SZ = 1 )
    if( temp < -32768 )
      CA ← 1
      RTA ← -32768
    else if( temp > 32767 )
      CA ← 1
      RTA ← 32767
    else
      CA ← 0
      RTA ← temp
  else
    if( temp < -128 )
      CA ← 1
      RTA ← -128
    else if( temp > 127 )
      CA ← 1
      RTA ← 127
    else
      CA ← 0
      RTA ← temp
else
  if( temp < 0 )
    CA ← 1
```

```
  RTA ← 0
else
  if( SZ = 1 )
    if( temp > 65535 )
      CA ← 1
      RTA ← 65535
    else
      CA ← 0
      RTA ← temp
  else
    if( temp > 255 )
      CA ← 1
      RTA ← 255
    else
      CA ← 0
      RTA ← temp
```

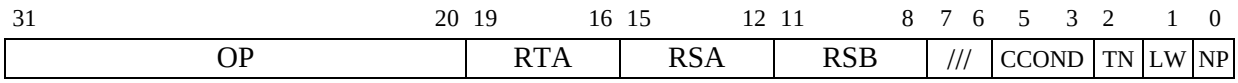
The general-purpose register RTA is set to the value of general-purpose register RSA, saturated to 16/8 bits based on the SZ bits using unsigned or signed saturation based on MD bit

In the assembler, the mnemonics `sat_md_h`/ `sat_md_b` are used for SZ=1/0. The mnemonics `sat_u_sz`/ `sat_s_sz` are used for MD=1/0.

4.3.12 Comparison and conditional selection instructions

4.3.12.1 Comparison

rc_lw_tn_ccond_np $rta,$rsa,$rsb




```
if ( TN = 0 )
{
  cc = [CCOND]1:0
  if( cc = 00b )
    cond ← [RSA] = [RSB]
  else if( cc = 01b )
    cond ← [RSA] < [RSB]
  else if( cc = 10b )
    cond ← [RSA] > [RSB]
  cond = cond ^ [CCOND]2
}
else
{
  cc = [CCOND]1:0
  if( cc = 00b )
    cond ← [RSA] = [RSB]
  else if( cc = 01b )
    cond ← [RSA] <u [RSB]
  else if( cc = 10b )
    cond ← [RSA] >u [RSB]
  cond = cond ^ [CCOND]2
}
CA ← cond
RTA = CA ? (NP ? 1 : -1) : 0;
```

The general-purpose register RTA is set to the comparison of the contents of general-purpose registers RSA and RSB and the carry bit. The carry bit is set to 1 if this comparison generates true, otherwise 0. In 32-bit architecture, the LW field is always 1(word).

If NP is 0 the result written into RTA is -1 if CA is set.

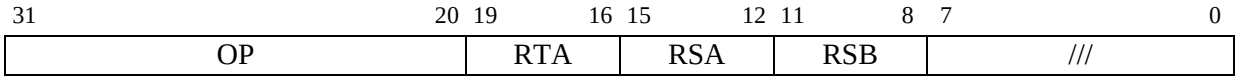
If NP is 1 the result written into RTA is 1 if CA is set.

Accumulate

In the assembler, the mnemonics `rc_w_s_ccond_np` / `rc_w_u_ccond_np` are used for TN=0/1. The mnemonics `rc_w_tn_eq_np` / `rc_w_tn_lt_np` / `rc_w_tn_gt_np` are used for CCOND=00 /01/11. The mnemonics `rc_w_tn_ccond_n` / `rc_w_tn_ccond_p` are used for NP=1/0.

4.3.12.2 Select

`sel $rta,$rsa,$rsb`

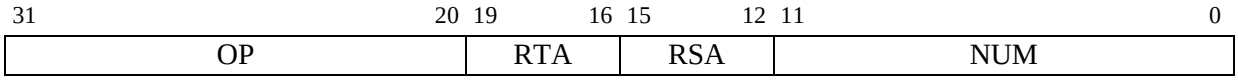



```
RTA ← if( CA = 1b ) [RSA] else [RSB]
```

Selection from RSA and RSB into RTA depending on current CA value.

4.3.12.3 Select immediate

`seli $rta,$rsa,num`



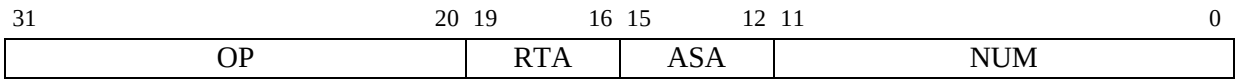

```
RTA ← if( CA = 1b ) [RSA] else NUM
```

Selection from RSA and the sign-extended value of the immediate NUM into RTA depending on current CA value.

4.3.13 Load/Store Instructions

4.3.13.1 Load Long

`ldil $rta,$asa,num`



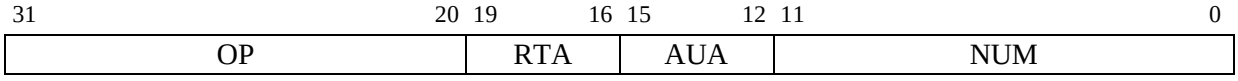

```
[RTA+1].[RTA] ← MEM( [ASA] + NUM )
```

Load the general-purpose register pair RTA with the 64-bit value stored in memory at the address in ASA incremented by the sign-extended value of the immediate NUM. RTA must be an even register address.

The results of this operation are undefined if the address is not 8B aligned.

4.3.13.2 Load long and Update

`ldul $rta,$aua,num`



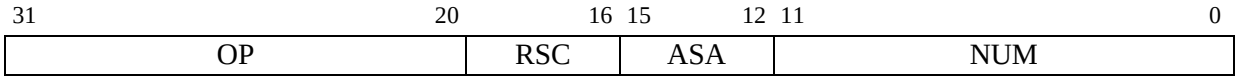

```
[RTA+1].[RTA] ← MEM( [AUA] )
AUA ← [AUA] + NUM
```

Load the general-purpose register pair RTA with the 64-bit value stored in memory at the address in AUA. Increment AUA by the sign-extended value of the NUM. RTA must be an even register address.

The results of this operation are undefined if the address is not 8B aligned.

4.3.13.3 Store Long

stil $rsc,$asa,num



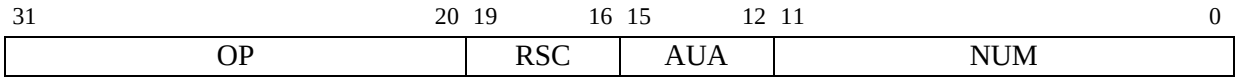

```
MEM( [ASA] + NUM ) ← [RSC+1].[RSC]
```

Store the 64-bit contents of general-purpose register pair RSC to the address in ASA incremented by the sign-extended value of the immediate NUM. RSC must be an even register address.

The results of this operation are undefined if the address is not 8B aligned.

4.3.13.4 Store long and Update

stul $rsc,$aua,num



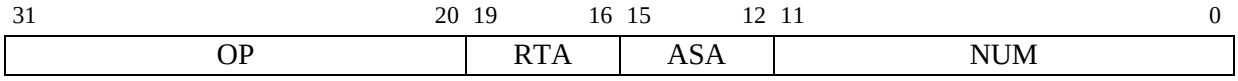

```
MEM( [AUA] ) ← [RSC+1].[RSC]
AUA ← [AUA] + NUM
```

Store the 64-bit contents of general-purpose register pair RSC to the address in AUA. Increment AUA by the sign-extended value of the NUM. RSC must be an even register address.

The results of this operation are undefined if the address is not 8B aligned.

4.3.13.5 Load word

`ldiw $rta,$asa,num`



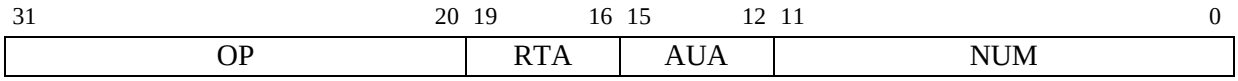

```
RTA ← MEM( [ASA] + NUM )
```

Load the general-purpose register RTA with the 32-bit value stored in memory at the address in ASA incremented by the sign-extended value of the immediate NUM.

The results of this operation are undefined if the address is not 4B aligned.

4.3.13.6 Load word and Update

`lduw $rta,$aua,num`



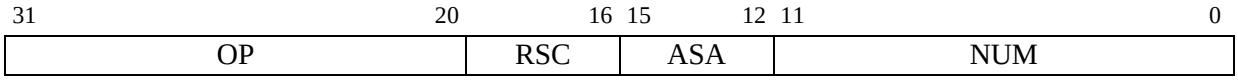

```
RTA ← MEM( [AUA] )
AUA ← [AUA] + NUM
```

Load the general-purpose register RTA with the 32-bit value stored in memory at the address in AUA. Increment AUA by the sign-extended value of the NUM.

The results of this operation are undefined if the address is not 4B aligned.

4.3.13.7 Store word

stiw $rsc,$asa,num



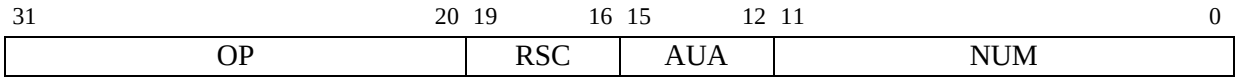

```
MEM( [ASA] + NUM ) ← [RSC]
```

Store the 32 bits contents of general-purpose register RSC to the memory at the address in ASA incremented by the sign-extended value of the immediate NUM.

The results of this operation are undefined if the address is not 4B aligned.

4.3.13.8 Store word and Update

stuw $rsc,$aua,num



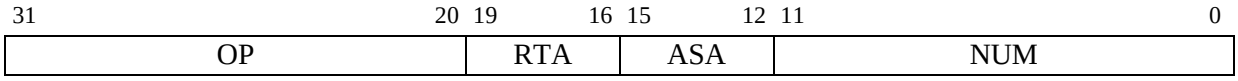

```
MEM( [AUA] ) ← [RSC]
AUA ← [AUA] + NUM
```

Store the 32 bits of contents of general-purpose register RSC to the address in AUA. Increment AUA by the sign-extended value of the NUM.

The results of this operation are undefined if the address is not 4B aligned.

4.3.13.9 Load Half-word

ldih $rta,$asa,num



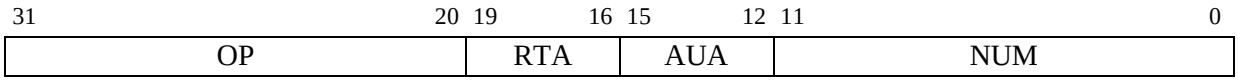

```
RTA ← MEM( [ASA] + NUM )
```

Load the general-purpose register RTA with the 16-bit value stored in memory at the address in ASA incremented by the sign-extended value of the immediate NUM, zero-extended to 32 bits.

The results of this operation are undefined if the address is not 2B aligned.

4.3.13.10 Load half-word and Update

lduh $rta,$aua,num



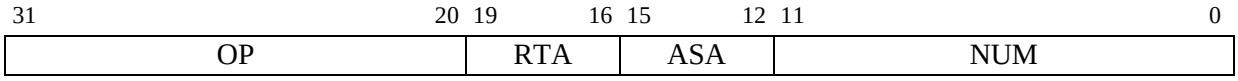

```
RTA ← MEM( [AUA] )
AUA ← [AUA] + NUM
```

Load the general-purpose register RTA with the 16-bit value stored in memory at the address in AUA, zero-extended to 32 bits. Increment AUA by the sign-extended value of the NUM.

The results of this operation are undefined if the address is not 2B aligned.

4.3.13.11 Load Signed Half-word

ldish $rta,$asa,num



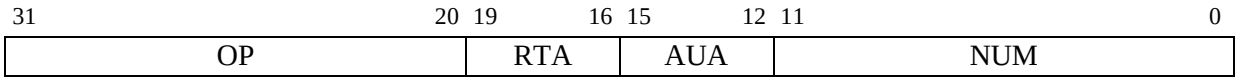

```
RTA ← MEM( [ASA] + NUM )
```

Load the general-purpose register RTA with the 16-bit value stored in memory at the address in ASA incremented by the sign-extended value of the immediate NUM, sign-extended to 32 bits.

The results of this operation are undefined if the address is not 2B aligned.

4.3.13.12 Load signed half-word and Update

ldush $rta,$aua,num



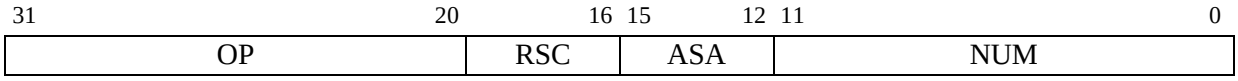

```
RTA ← MEM( [AUA] )
AUA ← [AUA] + NUM
```

Load the general-purpose register RTA with the 16-bit value stored in memory at the address in AUA, sign-extended to 32 bits. Increment AUA by the sign-extended value of the immediate NUM.

The results of this operation are undefined if the address is not 2B aligned.

4.3.13.13 Store Half-word

stih $rsc,$asa,num



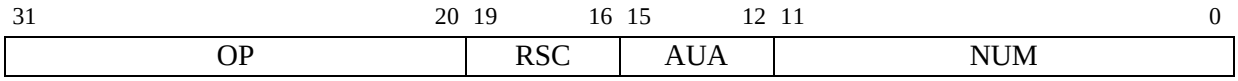
MEM([ASA] + NUM) ← $[RSC]_{15:0}$

Store the lower 16 bits of value stored in the general-purpose register RSC to the address in ASA incremented by the sign-extended value of the immediate NUM.

The results of this operation are undefined if the address is not 2B aligned.

4.3.13.14 Store half-word and Update

stuh $rsc,$asa,num



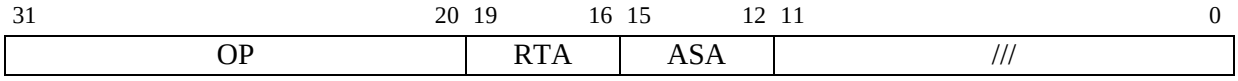
MEM([AUA]) ← $[RSC]_{15:0}$
AUA ← [AUA] + NUM

Store the lower 16 bits of value stored in the general-purpose register RSC to the address in AUA. Increment AUA by the sign-extended value of the NUM.

The results of this operation are undefined if the address is not 2B aligned.

4.3.13.15 Load Byte

ldib $rta,$asa,num

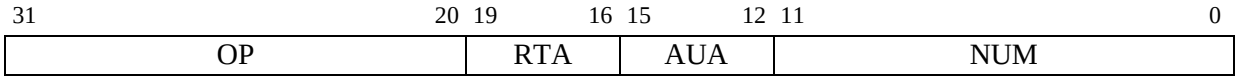



```
RTA ← MEM( [ASA] + NUM )
```

Load the general-purpose register RTA with the 8-bit value stored in memory at the address in ASA incremented by the sign-extended value of the immediate NUM, zero-extended to 32 bits.

4.3.13.16 Load byte and Update

ldub $rta,$aua,num

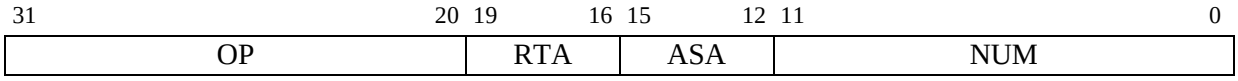



```
RTA ← MEM( [AUA] )
AUA ← [AUA] + NUM
```

Load the general-purpose register RTA with the 8-bit value stored in memory at the address in AUA, zero-extended to 32 bits. Increment AUA by the sign-extended value of the NUM value.

4.3.13.17 Load Signed Byte

ldisb $rta,$asa,num

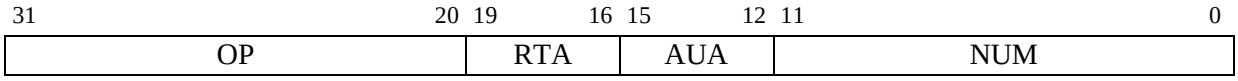



```
RTA ← MEM( [ASA] + NUM )
```

Load the general-purpose register RTA with the 8-bit value stored in memory at the address in ASA incremented by the sign-extended value of the immediate NUM, sign-extended to 32 bits.

4.3.13.18 Load signed byte and Update

ldusb $rta,$aua,num

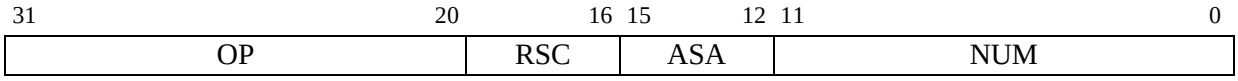



```
RTA ← MEM( [AUA] )
AUA ← [AUA] + NUM
```

Load the general-purpose register RTA with the 8-bit value stored in memory at the address in AUA, sign-extended to 32 bits. Increment AUA by the sign-extended value of the NUM value.

4.3.13.19 Store Byte

stib $rsc,$asa,num

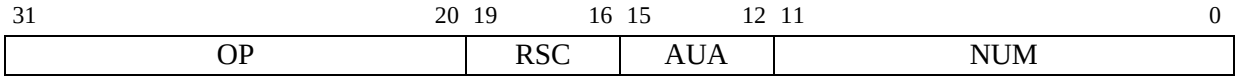


MEM([ASA] + NUM) ← $[RSC]_{7:0}$

Store the lower 8 bits of value stored in the general-purpose register RSC to the address in ASA incremented by the sign-extended value of the immediate NUM.

4.3.13.20 Store byte and Update

stub $rsc,$aua,num



MEM([AUA]) ← $[RSC]_{7:0}$
AUA ← [AUA] + NUM

Store the lower 8 bits of value stored in the general-purpose register RSC to the address in AUA. Increment AUA by the sign-extended value of the NUM.

5 Vector Unit

5.1 Overview

The vector unit extension specified in this chapter uses operations that perform on vectors fixed length。

The current version is designed to have vectors of length 16B, but the extension is designed to allow for future versions to have larger lengths, and be backward compatible.

This chapter describes the instructions and state used by the vector unit.

5.2 Vector Registers

5.2.1 Vector register

There are 16 16B registers, written as `$v0`...`$v15`. Future versions of the architecture may support longer vector lengths.

5.2.2 Vector Accumulator register

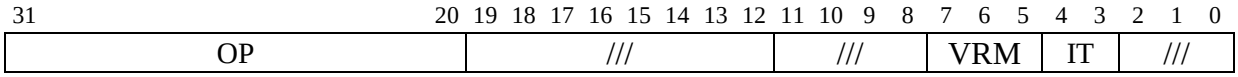
There are one 320-bit (use as 8 40bits or 16 20bits) vector accumulator register, call ACC.

5.2.3 Vector Column Accumulator register

There are one 196-bit (use as 8 8+16 bits or 16 4+8 bits) vector column accumulator register, it is a hidden register, call CACC. Use CACC register and VTA put together a complete column accumulator (8 40-bit or 16 20-bit).

5.3 Fields

5.3.1 Vector Rounding Model field



The vector rounding model is used to choose rounding model when calculated fix-point decimals multiply. When VRM(2) is 0, mean rounding model is from FIXCR register, else rounding model is decided by VRM(1:0).

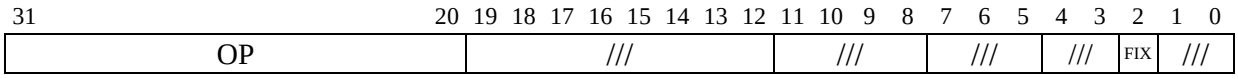
The 3 bits of the rounding mode control are interpreted as:

- 000: Use the VRM bits from the FIXCR Register
- 100: Round to nearest, half to even
- 101: Round to -∞
- 110: Round to +∞
- 111: Round to 0

In the assembler, the mnemonics `op_it_rdyn/ op_it_rzero/ op_it_rdown/ op_it_rup / op_it_rnear` are used for dynamic/nearest/zero/negative infinity/positive infinity (i.e. VRM = 0/4/5/6/7) variants of the operation *op* with size type *it*. If no variant is specified, round to nearest is assumed.

5.3.2 Fix-point decimals field

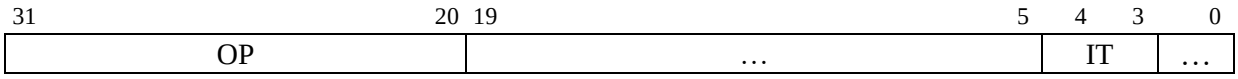
The vector fix-point decimals field is used with vector calculate type operations. It specifies the type of calculate that are operated on.



- 0: fix-point integer.
- 1: fix-point decimals.

5.3.3 Integer Element Type

The vector integer element type IT field is used with vector integer operations. It specifies the size of the elements that are operated on.



The bit encodings are:

- 00: byte.
- 01: half.
- 10: word.

Not all types are supported by all integer operations; unless otherwise specified, long is not supported.

In the assembler, the suffix _b/_h/_w is appended to the operation depending on IT=00/01/10.

5.4 Vector Instructions

In the following instruction description: VSA & VSB mean vector source register, VTA mean vector target register, VUD mean vector update register, VSD mean vector store destination register.

In the following instruction description use variable IL mean loop times, and variable EL mean elements length. $[VSA]_i$ mean i in elements. VSA_i mean i in bit.

5.4.1 Arithmetic Instructions

5.4.1.1 Vector Add

vadd_it_sat $vta,$vsa,$vsb

31 ... 20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5	4 3	2 1	0
OP	VTA	VSA	VSB	///	IT	///	SAT

```
if( IT = 10b )
  IL = 4
else if( IT = 01B)
  IL = 8
else
  IL = 16
for( i = 0; i < IL; i++ )
  v ← [VSA]i + [VSB]i
    if( SAT  = 0b )
      [VTA]i ← v
    else
      [VTA]i ← Saturate16(v)
```

Element-wise addition of the elements of vector registers VSA and VSB, writing the results to VTA, where the elements are integers of a size identified by IT [word or half or byte]. The value of the result may be saturated if SAT=1.

In the assembler, the mnemonics `vadd_it_lower`/ `vadd_it_sat` are used for SAT=0/1. The mnemonic `vadd_it` is equivalent to `vadd_it_lower`.

5.4.1.2 Vector Subtract

vsub_it_sat $vta,$vsa,$vsb

31 ... 20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5	4 3	2 1	0
OP	VTA	VSA	VSB	///	IT	///	SAT

```
If( IT = 10b )
  IL = 4
else if( IT = 01B)
  IL = 8
else
  IL = 16
for( i = 0; i < IL; i++ )
  v ← [VSA]i - [VSB]i
    if( SAT  = 0b )
      [VTA]i ← v
    else
      [VTA]i ← Saturate16(v)
```

Element-wise subtraction of the elements of vector registers VSA and VSB, writing the results to VTA, where the elements are integers of a size identified by IT [word or half or byte]. The value of the result may be saturated if SAT=1.

In the assembler, the mnemonics `vsub_it_lower`/ `vsub_it_sat` are used for SAT=0/1. The mnemonic `vsub_it` is equivalent to `vsub_it_lower`.

5.4.1.3 Vector Absolute Difference

`vdiff_it_sat $vta,$vsa,$vsb`

31 … 20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5	4 3	2 1	0
OP	VTA	VSA	VSB	///	IT	///	SAT

```
If( IT = 10b )
  IL = 4
else if( IT = 01B)
  IL = 8
else
  IL = 16
for( i = 0; i < IL; i++ )
  v ← |[VSA]i - [VSB]i|
    if( SAT  = 0b )
      [VTA]i ← v
    else
      [VTA]i ← Saturate16(v)
```

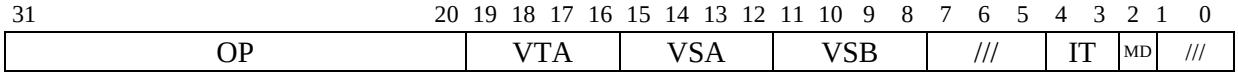
Element-wise absolute of the elements of vector registers VSA and VSB, writing the results to VTA, where the elements are integers of a size identified by IT [word or half or byte]. The value of the result may be saturated if SAT=1. In the unsaturated case the difference is an unsigned number.

In the assembler, the mnemonics `vdiff_it_lower`/ `vdiff_it_sat` are used for SAT=0/1. The mnemonic `vdiff_it` is equivalent to `vdiff_it_lower`.

5.4.2 Min/Max Instructions

5.4.2.1 Vector Minimum

vmin_it_md $vta,$vsa,$vsb



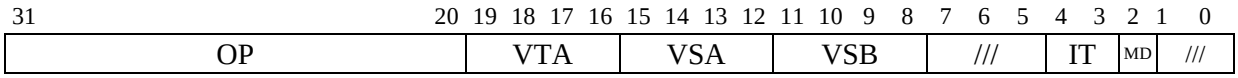

```
If( IT = 10b )
  IL = 4
else if( IT = 01B)
  IL = 8
else
  IL = 16
for( i = 0; i < IL; i++ )
  if( MD = 0b )
    [VTA]i ← if( [VSA]i < [VSB]i ) [VSA]i else [VSB]i
  else
    [VTA]i ← if( unsigned [VSA]i < Unsigned [VSB]i ) [VSA]i else [VSB]i
```

This instruction finds the smaller value for each element of vector registers VSA and VSB, and writes the results of each element to VTA, where the elements are assumed to be integers of a size identified by IT [word or half or byte]. The comparison will be signed or unsigned depending on MD.

In the assembler, the mnemonic `vmin_it_s`/ `vmin_it_u` are used for MD=0/1. `vmin_it` is equivalent to `vmin_it_s`.

5.4.2.2 Vector Maximum

vmax_it_md $vta,$vsa,$vsb




```
If( IT = 10b )
  IL = 4
else if( IT = 01B)
  IL = 8
else
  IL = 16
for( i = 0; i < IL; i++ )
  if( MD = 0b )
    [VTA]i ← if( [VSA]i > [VSB]i ) [VSA]i else [VSB]i
  else
    [VTA]i ← if( unsigned [VSA]i > Unsigned [VSB]i ) [VSA]i else [VSB]i
```

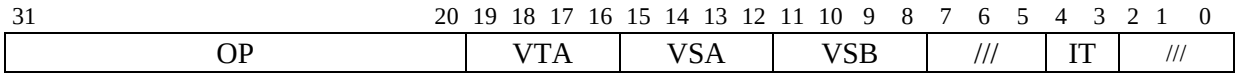
This instruction finds the larger value for each element of vector registers VSA and VSB, and writes the results of each element to VTA, where the elements are assumed to be integers of a size identified by IT field [word or half or byte]. The comparison will be signed or unsigned depending on MD field.

In the assembler, the mnemonic `vmax_it_s`/ `vmax_it_u` are used for MD=0/1. `vmax_it` is equivalent to `vmax_it_s`.

5.4.3 Logical Instructions

5.4.3.1 Vector And

vand_it $vta,$vsa,$vsb



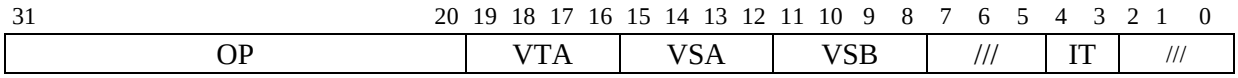

```
If( IT = 10b )
  IL = 4
else if( IT = 01B)
  IL = 8
else
  IL = 16
for( i = 0; i < IL; i++ )
   [VTA]ᵢ ← [VSA]ᵢ & [VSB]ᵢ
```

Element-wise bitwise AND the elements of vector registers VSA and VSB, writing the results to VTA, where the elements are integers of a size identified by IT [word or half or byte].

Especially, if the 2/4 adjacent bits is not all 1 when element length is half/word, that half/word of result will not be written in the destination register.

5.4.3.2 Vector Or

vor_it $vta,$vsa,$vsb



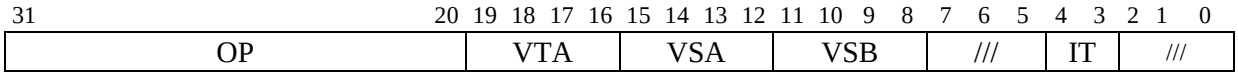

```
If( IT = 10b )
  IL = 4
else if( IT = 01B)
  IL = 8
else
  IL = 16
for( i = 0; i < IL; i++ )
```

```
[VTA]i ← [VSA]i | [VSB]i
```

Element-wise bitwise OR the elements of vector registers VSA and VSB, writing the results to VTA, where the elements are integers of a size identified by IT [word or half or byte].

5.4.3.3 Vector Cand

vcand_it $vta,$vsa,$vsb

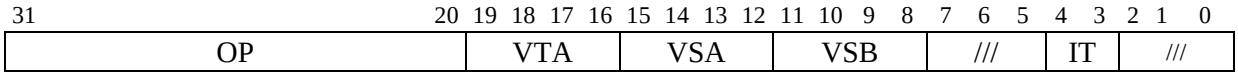



```
If( IT = 10b )
  IL = 4
else if( IT = 01B)
  IL = 8
else
  IL = 16
for( i = 0; i < IL; i++ )
  [VTA]i ← ~( [VSA]i ) & [VSB]i
```

Element-wise bitwise CAND the complement of the elements of vector registers VSA with the elements of VSB, writing the results to VTA, where the elements are integers of a size identified by IT [word or half or byte].

5.4.3.4 Vector Exclusive Or

vxor_it $vta,$vsa,$vsb



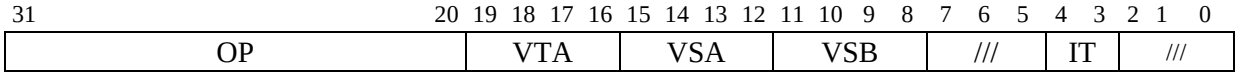

```
If( IT = 10b )
  IL = 4
else if( IT = 01B)
  IL = 8
else
  IL = 16
for( i = 0; i < IL; i++ )
  [VTA]i ← [VSA]i ^ [VSB]i
```

Element-wise bitwise XOR the elements of vector registers VSA and VSB, writing the results to VTA, where the elements are integers of a size identified by IT [word or half or byte].

5.4.4 Shift Instructions

5.4.4.1 Vector Shift Left

vshl_it $vta,$vsa,$vsb

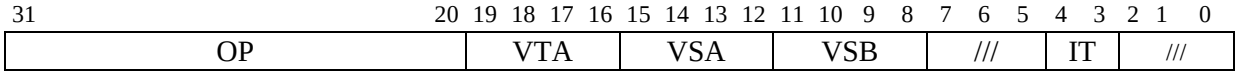



```
If( IT = 10b )
  IL = 4
else if( IT = 01B)
  IL = 8
else
  IL = 16
for( i = 0; i < IL; i++ )
  [VTA]i ← [VSA]i << ([VSB]i)2+IT:0
```

Element-wise left shift of the elements of vector registers VSA by a subset of bits of VSB, writing the results to VTA, where the elements are integers of a size identified by IT. The lowest 3/4/5 bits of VSB are used, depending on whether IT [word or half or byte].

5.4.4.2 Vector Shift Right

vshr_it $vta,$vsa,$vsb




```
If( IT = 10b )
  IL = 4
else if( IT = 01B)
  IL = 8
else
  IL = 16
for( i = 0; i < IL; i++ )
  [VTA]i ← [VSA]i >> ([VSB]i)2+IT:0
```

Element-wise logical right shift of the elements of vector registers VSA by a subset of bits of VSB, writing the results to VTA, where the elements are integers of a size identified by IT. The lowest 3/4/5 bits of VSB are used, depending on whether IT [word or half or byte]. This is a logical right shift; i.e. zeroes are shifted in.

5.4.4.3 Vector Shift Right Arithmetic

vshra_it $vta,$vsa,$vsb

31 20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5	4 3	2 1 0
OP	VTA	VSA	VSB	///	IT	///

```
If( IT = 10b )
  IL = 4
else if( IT = 01B)
  IL = 8
else
  IL = 16
for( i = 0; i < IL; i++ )
  [VTA]i ← [VSA]i >>arith ([VSB]i)2+IT:0
```

Element-wise arithmetic right shift of the elements of vector registers VSA by a subset of bits of VSB, writing the results to VTA, where the elements are integers of a size identified by IT. The lowest 3/4/5 bits of VSB are used, depending on whether IT [word or half or byte]. This is an arithmetic right shift; i.e. the sign bit is shifted in.

5.4.5 Multiply Instructions

5.4.5.1 Vector Multiply

vmul_it_sat_fix_vrm $vta,$vsa,$vsb

31 20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5	4 3	2	1	0
OP	VTA	VSA	VSB	VRM	IT	FIX	/	SAT

```
If( IT = 01b )
  IL = 8; EL = 16
else
  IL = 16; EL = 8
for( i = 0; i < IL; i++ )
  v ← [VSA]i * [VSB]i
  if( SAT  = 0b )
    if( FIX = 0b )
      [VTA]i ← v(EL-1:0)
    else
      [VTA]i ← v(2*EL-1:EL)
  else
    if( FIX = 0b )
      [VTA]i ← SaturateEL(v>>EL-1)
    else
      [VTA]i ← SaturateEL(v>>2*EL-2)
  if( v = 0 )
    [CACC]i ← 0
```

Element-wise multiplication of the elements of vector registers VSA and VSB, truncate/saturate to the size of the integer identified by IT, writing the results to VTA. Where the elements are integers of a size identified by IT [half or byte]. The value of the result may be saturated if SAT=1.

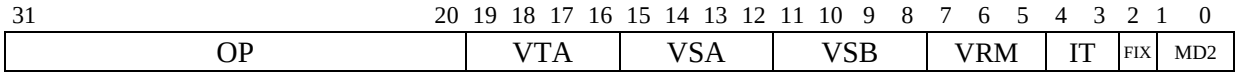
This instruction supports fixed-point decimal when FIX = 1, and the Rounding mode identified by VRM field/FIXCR register.

If the result of vector register VSA and VSB is zero, we think of it as a signal to prepare for column accumulate, then this instruction will reset CACC register to all zero.

In the assembler, the mnemonics `vmul_it_lower`/ `vmul_it_sat` are used for SAT=0/1. The mnemonic `vmul_it` is equivalent to `vmul_it_lower`.

5.4.5.2 Vector Multiply Signed Double-width result

v2mul_it_md2_fix_vrm $vta,$vsa,$vsb



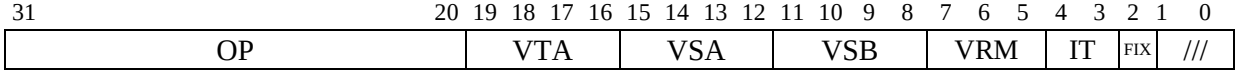

```
If( IT = 01b )
  IL = 8
else
  IL = 16
for( i = 0; i < IL; i++ )
  if( FIX = 0b )
    if( MD2 = 00b )
      [VTA]i ← Signed([VSA]i) * Signed([VSB]i)
    else if( MD2 = 01b )
      [VTA]i ← Signed([VSA]i) * Unsigned([VSB]i)
    else if( MD2 = 10b )
      [VTA]i ← Unsigned([VSA]i) * Signed([VSB]i)
    else if( MD2 = 11b )
      [VTA]i ← Unsigned([VSA]i) * Unsigned([VSB]i)
  if( FIX = 1b )
    [VTA]i ← Fixed([VSA]i) * Fixed([VSB]i)
```

Element-wise multiplication of the elements of vector registers VSA and VSB assuming both are signed, writing the results to $[VTA]_i$ and $[VTA]_{i+1}$. The elements of the inputs are integers of a size identified by IT [half or byte], while the result is twice that size [word or half].

This instruction result is fixed-point decimal when FIX field is 1. The rounding mode is given by VRM field/FIXCR register.

5.4.5.3 Vector Multiply Saturating Double-width result

v2muls_it_fix_vrm $vta,$vsa,$vsb



```
If( IT = 01b )
  IL = 8
else
  IL = 16
for( i = 0; i < IL; i++ )
    [VTA]i← Saturate(IT+1)( [VSA]i * [VSB]i * 2 )
```

Element-wise multiplication of the elements of vector registers VSA and VSB, multiplying by 2, and saturating, writing the results to $[VTA]_i$ and $[VTA]_{i+1}$. The elements of the inputs are integers of a size identified by IT [half or byte], while the result is twice that size [word or half].

This instruction result is fixed-point decimal when FIX field is 1. The rounding mode is given by VRM field/FIXCR register.

5.4.6 Vector Compare Instructions

5.4.6.1 Vector Compare Signed

vcmps_it_vcond $vta,$vsa,$vsb

31 20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5	4 3	2 1 0
OP	VTA	VSA	VSB	///	IT	VCOND

```
If( IT = 10b )
  IL = 4
  EL = 32
else if( IT = 01b)
  IL = 8
  EL = 16
else
  IL = 16
  EL = 8
for( i = 0; i < IL; i++ )
  cc = [VCOND]1:0
  if( cc = 00b )
    VTAi*EL = [VSA]i = [VSB]i
  else if( cc = 01b )
    VTAi*EL = [VSA]i < [VSB]i
  else if( cc = 10b )
    VTAi*EL = [VSA]i > [VSB]i
  cond = cond ^ [VCOND]2
```

Element-wise comparison the elements of vector registers VSA and VSB for the condition specified by VCOND, writing the results to vector register VTA, where the input elements are treated as signed integers of a size identified by IT [word or half or byte].

In the assembler, the mnemonics `vcmps_it_eq`/ `vcmps_it_lt`/ `vcmps_it_gt`/ `vcmps_it_ne`/ `vcmps_it_ge`/ `vcmps_it_le` are used for VCOND=0/1/2/4/5/6.

5.4.6.2 Vector Compare Unsigned

vcmpu_it_vcond $vta,$vsa,$vsb

31 ... 20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5	4 3	2 1 0
OP	VTA	VSA	VSB	///	IT	VCOND

```
If( IT = 10b )
  IL = 4
  EL = 32
else if( IT = 01b)
  IL = 8
  EL = 16
else
  IL = 16
  EL = 8
for( i = 0; i < IL; i++ )
  cc = [VCOND]1:0
  if( cc = 00b )
    VTAi*EL = [VSA]i = [VSB]i
  else if( cc = 01b )
    VTAi*EL = [VSA]i < [VSB]i
  else if( cc = 10b )
    VTAi*EL = [VSA]i > [VSB]i
  cond = cond ^ [VCOND]2
```

Element-wise comparison the elements of vector registers VSA and VSB for the condition specified by VCOND, writing the results to vector register VTA, where the input elements are treated as signed integers of a size identified by IT [word or half or byte].

In the assembler, the mnemonics `vcmpu_it_eq`/ `vcmpu_it_lt`/ `vcmpu_it_gt`/ `vcmpu_it_ne`/ `vcmpu_it_ge`/ `vcmpu_it_le` are used for VCOND=0/1/2/4/5/6.

5.4.6.3 Vector Select

vsel_it $vud,$vsa,$vsb

31 ... 20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5	4 3	2 1 0
OP	VUD	VSA	VSB	///	IT	///

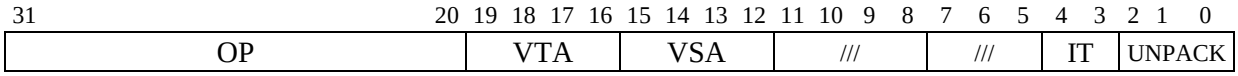
```
If( IT = 10b )
  IL = 4
  EL = 8
else if( IT = 01B)
  IL = 8
  EL = 16
else
  IL = 16
  EL = 32
for( i = 0; i < IL; i++ )
  VUDi ← if( VSBi*EL = 1b ) [VSA]i else [VUD]i
```

Element-wise selection between the elements of vector registers VUD and VSA, depending on whether the corresponding bit of the vector register VSB is true or false, and write the results to VUD, where the elements are integers of a size identified by IT [word or half or byte].

5.4.7 Vector Conversion Instructions

5.4.7.1 Vector Unpack

vunpack_it_unpack $vta,$vsa



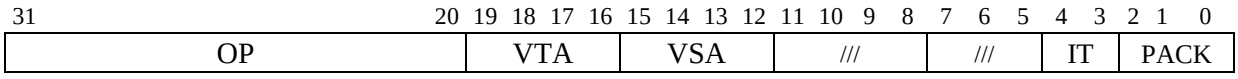

```
If( IT = 01b )
  IL = 8
else
  IL = 16
for( i = 0; i < IL; i++ )
  if( PACK = 000b )
    [VTA+1]i,[VTA]i ← SignExtendIT+1( [VSA]i )
  else if( PACK = 001b )
    [VTA+1]i,[VTA]i ← ZeroExtendIT+1( [VSA]i )
  else if( PACK = 100b )
    [VTA+1]i,[VTA]i ← ZeroPadIT+1( [VSA]i )
```

Copy the elements of vector registers VSA to vector register VTA, where the elements are integers of size identified by IT [byte or half], and are extended to twice the width prior to writing to VTA. The extension is controlled by the PACK field.

In the assembler, the mnemonics `vunpack_it_sign`/ `vunpack_it_zero`/ `vunpack_it_upper` is used for UNPACK=0/1/4.

5.4.7.2 Vector Pack

vpack_it_pack $vta,$vsa




```
If( IT = 10b )
  IL = 4
else
  IL = 8
for( i = 0; i < IL; i++ )
  if( PACK = 000b )
```

```
    [VTA]i ← LowerIT-1( [VSA]i )
  else if( PACK = 001b )
    [VTA]i ← SaturateIT-1( [VSA]i )
  else if( PACK = 010b )
    [VTA]i ← SaturateUnsignedIT-1( [VSA]i )
  else if( PACK = 011b )
    [VTA]i ← SaturateClipNegativeIT-1( [VSA]i )
  else if( PACK == 100b )
    [VTA]i ← UpperRoundToNearestEvenIT-1( [VSA]i )
  else if( PACK == 101b )
    [VTA]i ← UpperRoundToZeroIT-1( [VSA]i )
  else if( PACK == 110b )
    [VTA]i ← UpperIT-1( [VSA]i ) // Same as UpperRoundDown
  else if( PACK == 111b )
    [VTA]i ← UpperRoundUpIT-1( [VSA]i )
```

Copy the elements of vector registers VSA to vector register VTA, where the elements are integers of size identified by IT [word or half], and are converted to half the width prior to writing to VTA. The method by which they are converted is controlled by the PACK field.

In the assembler, the mnemonics `vpack_it_lower/ vpack_it_sat/ vpack_it_lsatu/ vpack_it_satc/ vpack_it_rnear/ vpack_it_rzero/ vpack_it_rdown/ vpack_it_up` to indicated PACK=0/1/2/3/4/5/6/7.

5.4.8 Vector Element Instructions

5.4.8.1 Vector Broadcast

vbrd_it $vta,$rsa

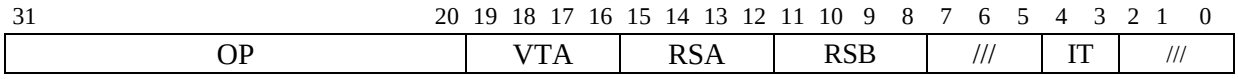
31 — 20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5	4 3	2 1 0
OP	VTA	RSA	///	///	IT	///

```
If( IT = 10b )
  IL = 4
  EL = 32
else if( IT = 01B)
  IL = 8
  EL = 16
else
  IL = 16
  EL = 8
for( i = 0; i < IL; i++ )
  [VTA]i ← RSAEL:0
```

Copy the value in general-purpose register RSA to the elements of the vector register VTA. The value is truncated to match the size of the elements as identified by IT [word or half or byte].

5.4.8.2 Vector Set from GPR

vsetr_it $vta,$rsa,$rsb



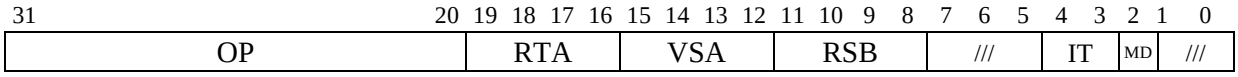
VTA[RSB] ← [RSA]

Copy the value in general-purpose register RSA to the position in vector register VTA identified by register RSB. The value is truncated to match the size of the elements as identified by IT.

In the assembler, the mnemonics `vsetr_it`.

5.4.8.3 GPR Set from Vector

rsetv_it_md $rta,$vsa,$rsb




```
if( MD = 0b )
  RTA ← SignExtendIT( [VSA][RSB] )
else
  RTA ← ZeroExtendIT( [VSA][RSB] )
```

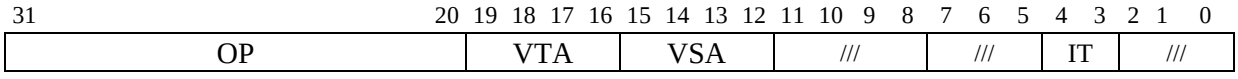
Copy the element value in vector register VSA at the position identified by register RSB to the general-purpose register RTA. The value is assumed to be an integer of size identified by IT[word or byte, half], and is sign/zero-extended to 64-bits based on the MD field.

In the assembler, the mnemonics `rsetv_it_s`/ `rsetv_it_u` are used for MD=0/1.

5.4.9 Vector Reorder Instructions

5.4.9.1 Vector Copy

vcopy_it $vta,$vsa



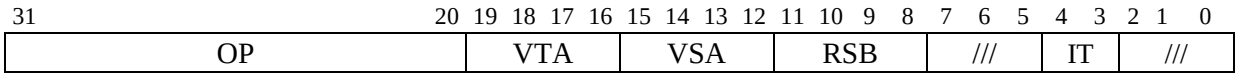

```
If( IT = 10b )
  IL = 4
else if( IT = 01b )
```

```
  IL = 8
else
  IL = 16
for( i = 0; i < IL; i++ )
  [VTA]i ← [VSA]i
```

Copy elements of vector registers VSA to vector register VTA. The elements are integers of size given by IT [word or byte, half].

5.4.9.2 Vector Tail

vtail_it $vta,$vsa,$rsb

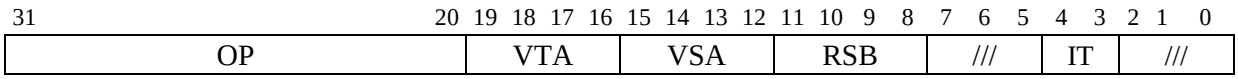



```
If( IT = 01b )
  IL = 8
else
  IL = 16
for( j = 0, i = [RSB]; j < IL; I < IL; i ++, j++ )
  [VTA]j ← [VSA]i
```

Copy a subset of the elements of vector registers VSA to vector register VTA, where the starting position is stored in register RSB. The elements are integers of size given by IT[byte or half].

5.4.9.3 Vector Spread

vspread_it $vta,$vsa,$rsb

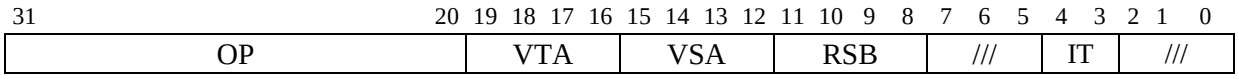



```
n = 0
If( IT = 01b )
  IL = 8
else
  IL = 16
for(i = 0; i < (IL/[RSB]); i ++ )
  for(j = 0; j < [RSB]; j ++, n++ )
    [VTA]n ← [VSA]i
```

Copy (IL/ [RSB]) elements of vector registers VSA to vector register VTA, replicating each element RSB times. When [RSB] is 0, this instruction acts the same effect as NOP. The elements are integers of size given by IT [half or byte].

5.4.9.4 Vector Duplicate

vdupl_it $vta,$vsa,$rsb



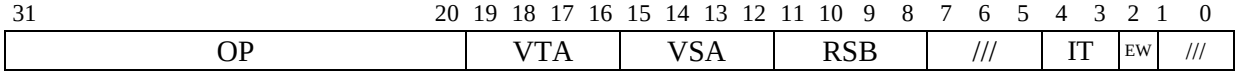

```
If( IT = 01b )
  IL = 8
else
  IL = 16
n = 0
for(i = 0; i < (IL/[RSB]); i ++ )
  for(j = 0; j < [RSB]; j ++, n++ )
    [VTA]n ← [VSA]j
```

Make (IL/ [RSB]) copies of the [RSB] elements of vector registers VSA to vector register VTA. The elements are integers of size given by IT[byte or half].

Especially, if the 2/4 adjacent bits is not all 1 when element length is half/word, that half/word of result will not be written in the destination register.

5.4.9.5 Vector Extract

vext_it_ew $vta,$vsa,$rsb



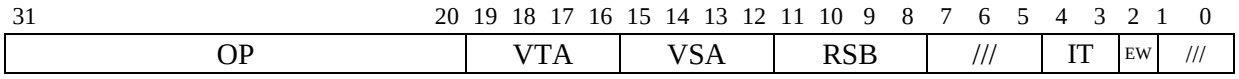

```
If( IT = 01b )
  IL = 8
else
  IL = 16
if( EW = 0b )
  n = 0
else
  n = [RSB] – 1
for(i = 0; i < (IL/[RSB]); i ++, n += [RSB] )
  if( ew = 0 )
    [VTA]i ← [VSA]n
```

Treat the vector VSA as (IL/ [RSB]) groups of NSB elements. From each of the groups, select either the beginning or ending element of the group, and copy to VTA. The beginning/ending selection is controlled by the EW field. The elements are integers of size given by IT[byte or half]. When [RSB] is 0, this instruction acts the same effect as NOP.

In the assembler, the mnemonics `vext_it_b`/ `vext_it_e` are used for EW=0/1.

5.4.9.6 Vector Extract But

vextb_it_ew $vta,$vsa,$rsb



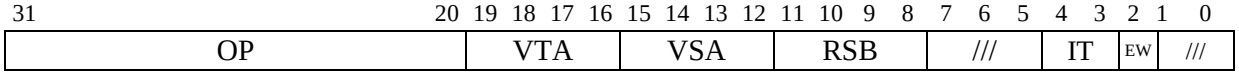

```
If( IT = 01b )
  IL = 8
else
  IL = 16
if( EW = 0b )
  f = 0
else
  f = [RSB]-1
m = 0
n = 0
for(i = 0; i < (IL/[RSB]); i ++ )
  for(j = 0; j < [RSB]; j ++, n++ )
    if( j != f )
      [VTA]m ← [VSA]n
      m++
```

Treat the vector VSA as (IL/ [RSB]) groups of NSB elements. From each of the groups, select all but either the beginning or ending element of the group, and copy to VTA; thus [RSB] elements from each of the NSA groups are copied to VTA. The beginning/ending selection is controlled by the EW field. The elements are integers of size given by IT[byte or half]. When [RSB] is 0/1, this instruction acts the same effect as NOP.

In the assembler, the mnemonics `vextb_it_b`/ `vextb_it_e` are used for EW=0/1.

5.4.9.7 Vector De-interleave

vdilv_it_ew $vta,$vsa,$rsb



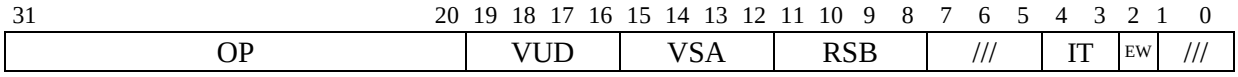

```
If( IT = 01b )
  IL = 8
else
  IL = 16
if( EW = 0b )
  n = 0
else
  n = [RSB]
for(i = 0; i < (IL/[RSB]); i ++ )
  for(j = 0; j < [RSB]; j ++, m++, n++ )
    [VTA]m ← [VSA]n
  n++
```

Treat the vector VSA as (IL/ [RSB]) pairs of groups of RSB elements. From each pair of groups, copy one of the groups to VTA. The first/second selection is controlled by the EW field. The elements are integers of size given by IT[byte or half]. When [RSB] is 0, this instruction acts the same effect as NOP.

In the assembler, the mnemonics `vdilv_it_b`/ `vdilv_it_e` are used for EW=0/1

5.4.9.8 Vector Concatenate

`vconcat_it_ew $vud,$vsa,$rsb`




```
If( IT = 01b )
  IL = 8
else
  IL = 16
if (EW = 0)
  for( i = 0; i < IL; i ++ )
    if( i < [RSB] )
      [VUD]i ← [VUD]i
    else
      [VUD]i ← [VSA]i-[RSB]
else
  for( i = 0; i < IL-[RSB]; i ++ )
    [VUD]i ← [VUD][RSB]+i
  for( m = 0; i = TL-[RSB]; i < IL; i ++; m ++ )
    [VUD]i ← [VSA]m
```

This instruction had two mode, decide by EW field:

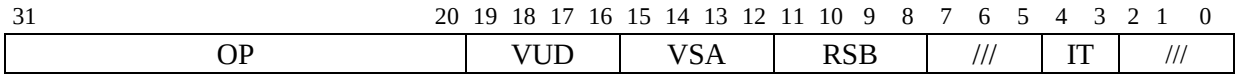
0: Copy elements of vector registers VUD to vector register VUD, where the number of elements to be copied is [RSB]. Then copy the rest of the values from vector register VSA, starting at the beginning of VSA. Continue copying till the total number of values copied to VUD from VUD and VSA is the IL number.

1: Copy elements of vector register VUD to vector register VUD, where the number of elements to be copied is IL-[RSB], starting at [VUD] [RSB]. Then copy the rest of the values from vector register VSA, starting at the beginning of VSA. Continue copying till the total number of values copied to VUD from VUD and VSA is the IL number.

The elements are integers of size given by IT [half or byte]. In the assembler, the mnemonics `vconcat_it_b`/ `vconcat_it_e` are used for EW=0/1

5.4.9.9 Vector Insert

vins_it $vud,$vsa,$rsb

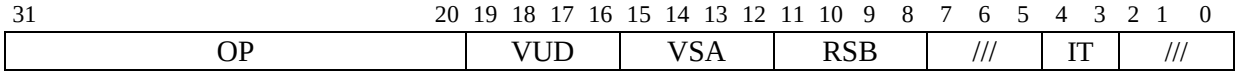



```
If( IT = 01b )
  IL = 8
else
  IL = 16
for( i = 0; m = 0; i < IL; i ++ )
  if( i = [RSB] )
    [VUD]i = [VSA]0
  else
    [VUD]i = [VUD]m
    m ++
```

Copy elements of vector register VUD to VUD, where the number of elements to be copied is [RSB], then followed by an element of vector register VSA, and then copy the rest of value form vector register VUD. The elements are integers of size given by IT[byte or half].

5.4.9.10 Vector Interleave

vilv_it $vud,$vsa,$rsb



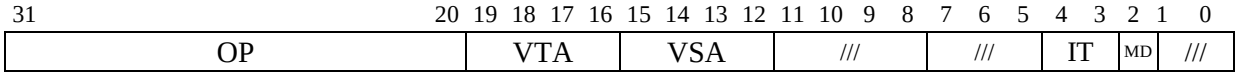

```
n = 0
m = 0
k = 0
If( IT = 01b )
  IL = 8
else
  IL = 16
for( i = 0; i < (IL/2*[RSB]); i ++ )
  for(j = 0; j < [RSB]; j ++, n++, m++ )
    [VUD]n ← [VUD]m
  for(j = 0; j < [RSB]; j ++, n++, k++ )
    [VUD]n ← [VSA]k
```

Copy elements of vector register VUD to VUD, where the number of elements to be copied is [RSB], then also copy [RSB] elements of vector register VSA to VUD, and repeat (IL/ 2*[RSB]) times. The elements are integers of size given by IT[byte or half].

5.4.10 Vector Search Instructions

5.4.10.1 Vector Min Across

vmina_it_md $vta,$vsa



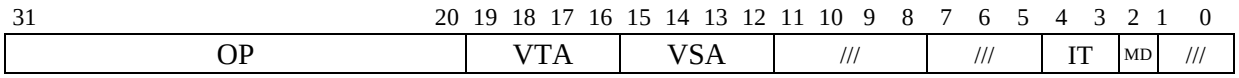

```
v ← [VSA]0
If( IT = 01b )
  IL = 8
else
  IL = 16
for( j = 0; j < IL; j++ )
  if( MD = 0b )
    v = min_signed(v, [VSA]j)
  else
    v = min_unsigned(v, [VSA]j)
[VTA]0 ← v
```

Treat the vector VSA of elements. From the groups, find the minimum value, and copy to VTA. The minimum is determined using signed/unsigned comparisons based on the value of MD. The elements are integers of size given by IT[byte or half].

In the assembler, the mnemonics `vmina_it_s`/ `vmina_it_u` are used for MD=0/1. The mnemonic `vmina_it` is equivalent to `vmina_it_s`.

5.4.10.2 Vector Max Across

vmaxa_it_md $vta,$vsa




```
v ← [VSA]0
If( IT = 1b )
  IL = 8
else
  IL = 16
for( j = 0; j < IL; j++ )
  if( MD = 0b )
    v = max_signed(v, [VSA]j)
  else
    v = max_unsigned(v, [VSA]j)
[VTA]0 ← v
```

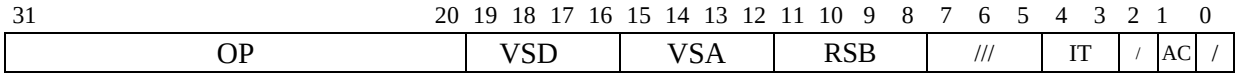
Treat the vector VSA elements. From the groups, find the maximum value, and copy to VTA. The maximum is determined using signed/unsigned comparisons based on the value of MD. The elements are integers of size given by IT[byte or half].

In the assembler, the mnemonics `vmaxa_it_s`/ `vmaxa_it_u` are used for MD=0/1. The mnemonic `vmaxa_it` is equivalent to `vmaxa_it_s`.

5.4.11 Vector Filter Instructions

5.4.11.1 Vector Filter Accumulator

`vfiltc_it_ac $vsd,$vsa,$rsb`




```
if( IT = 01b )
  IL = 8
else
  IL = 16
VWD = VSA
for( i = 0; i < [RSB]; i ++ )
  SUM = AC ? [ACC]i : 0;
  for( j = 0; j < IL; j ++ )
    SUM = SUM + [VWD]j * [VSD]j
  VWD = VWD127:IL-1 . (VSA+1)IL-1:0
  [ACC]i = Saturate( SUM )
VSA = VWD
```

vfiltc is a multicycle instruction of which the latency depends on repeat times given by RSB area, convolve the element of VSA against element of VSD, and write the saturated results to ACC register. Initialize sum to ACC register or zero according to AC field, the elements are signed integers of size given by IT field.

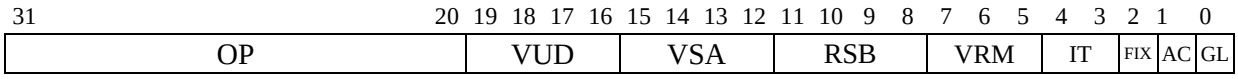
VSA field indicates the vector register couple (even-odd) including VSA and VSA+1, VSA records the filter window and VSA+1 records the next several elements for window to shift in.

At the beginning, VWD equals to VSA, after each convolve repeat, VWD is updated by shifting out the first element of VWD and shifting in the first element of VSA+1.

In the assembler, the mnemonics `vfiltc_it_zr`/ `vfiltc_it_ac` are used for AC=0/1. mnemonics `vfiltc_h_ac/vfiltc_b_ac` are used for IT=01/00.

5.4.11.2 Vector Filter

vfiltv_it_vrm_fix_ac_gl $vud,$vsa,$rsb




```
if( IT = 01b )
  IL = 8
else
  IL = 16
VWD = VSA
for( i = 0; i < [RSB]; i ++ )
  if( AC = 1 )
    SUM = [ACC]i
  else
    SUM = 0
  for( j = 0; j < IL; j ++ )
    SUM = SUM + [VWD]j * [VUD]j
  VWD = VWD127:IL-1 . (VSA+1)IL-1:0
  if( fix = 1 )
    if( GL = 1 )
       SUM = RoundingIT*2( SUM )
       [VUD+1]i = SaturateIT*2( SUM )
    else
       SUM = RoundingIT( SUM )
       [VUD+1]i = SaturateIT( SUM )
  else
    if( GL = 1 )
       [VUD+1]i = SaturateIT*2( SUM )
    else
       [VUD+1]i = SaturateIT( SUM )
VSA = VWD
```

vfiltv is a multicycle instruction of which the latency depends on repeat times given by RSB area, convolve the element of VSA against element of VUD, and write the saturated results to VUD+1 register. Initialize sum to ACC register or zero according to AC field, the elements are signed integer or signed fix-point decimal given by FIX field and size given by IT field, GL field decide the size of result is double or not.

VSA field indicates the vector register couple (even-odd) including VSA and VSA+1, VSA records the filter window and VSA+1 records the next several elements for window to shift in. VUD field also indicates the vector register couple like VSA, VUD will join calculation as another resource, VUD+1 is the target vector register.

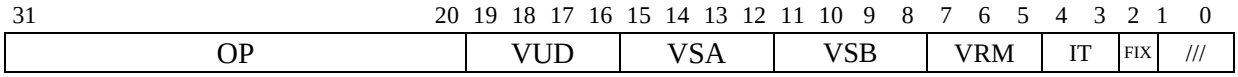
At the beginning, VWD equals to VSA, after each convolve repeat, VWD is updated by shifting out the first element of VWD and shifting in the first element of VSA+1.

This instruction supports fixed-point decimal when FIX field = 1, and the rounding mode identified by VRM field/FIXCR register.

In the assembler, the mnemonics `vfiltv_zr`/ `vfiltv_ac` are used for AC=0/1. The mnemonics *`vfiltv_h/vfiltv_b`* are used for IT=01/00. The mnemonics *`vfiltv_int/vfilt_fix`* are used for FIX=0/1. The mnemonics `vfiltv_sl/vfiltv_db` are used for GL=0/1.

5.4.11.3 Vector Column Multiply Sum

`vcsum_it_fix $vud,$vsa,$vsb`



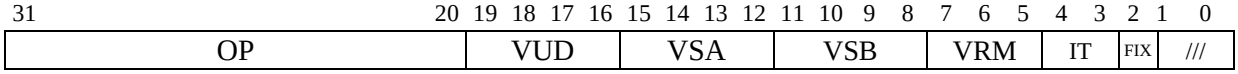

```
If( IT = 01b )
  IL = 8 EL = 16
else
  IL = 16 EL = 8
for( j = 0; j < IL; j++)
  [VTA]i,[CACC]i = [CACC]i.[VUD]i + [VSA]j * [VSB]j
```

Multiply the corresponding elements of VSA and VSB together and sum every product with the accumulator register CACC. Write the low EL bits of sum into VUD, write the rest bits to CACC. The elements are signed integers of a size identified by IT [half or byte].

This instruction result is fixed-point decimal when FIX field is 1. The rounding mode is given by VRM field/FIXCR register.

5.4.11.4 Vector Column Multiply Sum Double-width

`v2csum_it_fix $vud,$vsa,$vsb`




```
If( IT = 01b )
  IL = 8
else
  IL = 16
for( j = 0; j < IL; j++)
  [VUD]i+1 & [VUD]i = [CACC]i.[VUD]i + [VSA]j * [VSB]j
```

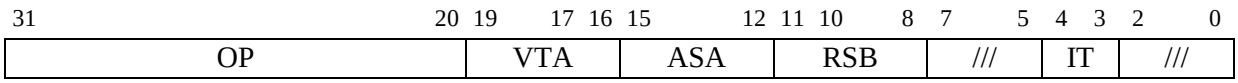
Multiply the corresponding elements of VSA and VSB together and sum every product with the accumulator register CACC. write full saturated sum into $[VUD]_i$ and $[VUD]_{i+1}$. The elements are signed integers of a size identified by IT [half or byte].

This instruction result is fixed-point decimal when FIX field is 1. The rounding mode is given by VRM field/FIXCR register.

5.4.12 Vector Load/Store Instructions

5.4.12.1 Load Vector

Ldv $vta,$asa,$rsb

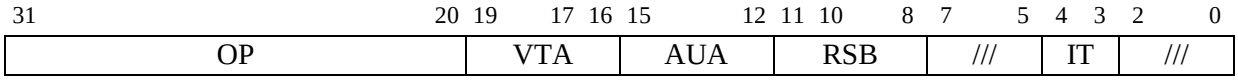



```
addr = [ASA]
for( i = 0; i < [RSB]; i ++ )
  VTAi ← MEM( addr )
  addr += bytesIT
```

Load the elements of vector register VTA with values stored in memory at the addresses starting at ASA.

5.4.12.2 Load Vector and Update

Lduv $vta,$aua,$rsb

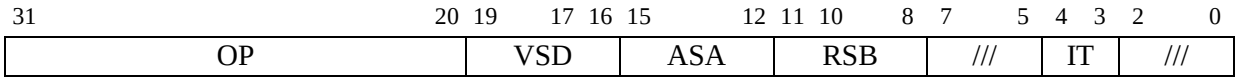



```
addr = [AUA]
for( i = 0; i < [RSB]; i ++ )
  VTAi ← MEM( addr )
   addr += bytesIT
AUA ← addr
```

Load the elements of vector register VTA with values stored in memory at the addresses starting at AUA.

5.4.12.3 Store Vector

Stv $vsd,$asa,$rsb

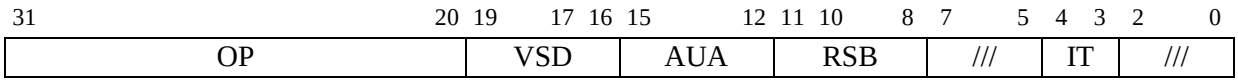



```
addr = [ASA]
for( i = 0; i < [RSB]; i ++ )
  MEM( addr ) ← VSD
  addr += byteIT
```

Store the elements of vector register VSD to memory at the addresses starting at ASA.

5.4.12.4 Store Vector and Update

Stuv $vsd,$aua,$rsb



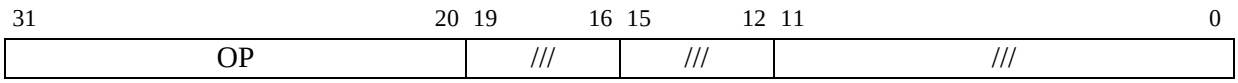

```
addr = [AUA]
for( i = 0; i < [RSB]; i ++ )
  MEM( addr ) ← VSDi
   addr += bytesIT
AUA ← addr
```

Store the elements of vector register VSD to memory at the addresses starting at AUA.

6 Special Purpose Instructions

6.1 No-op

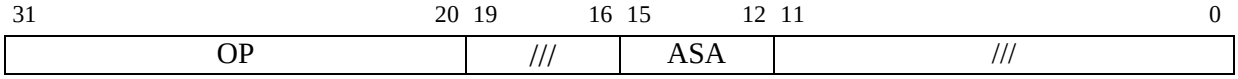
nop



Do nothing.

6.2 Pre-fetch

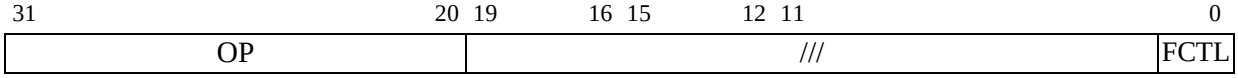
Prefetch $asa



If the line is not present, fetch it. If the line is already present or the address is uncacheable, this is a nop. It expects that it has read permissions to the line.

6.3 Data Fence

Fence_fctl

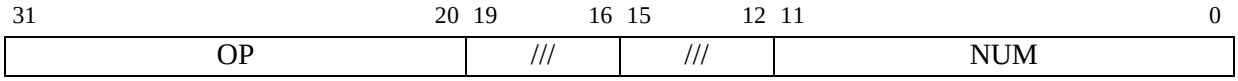


This instruction acts as a fence between store (containing cast-out) and following load or store. If FCTL is 01, fence following load, 10 fence following store, 11 fence following load or store. The following load or store must wait until the store data is written to memory.

In the assembler, the mnemonics `fence_load/fence_store/fence_both` are used for FCTL=1/2/3. Fence is equivalent to `fence_both`.

6.4 Trap

`trap num`

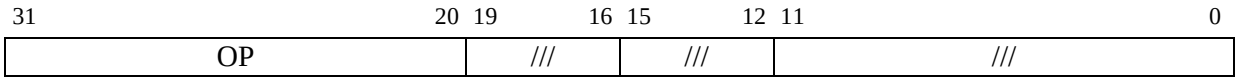


Control is transferred to the trap-handler identified by the NUM. The behavior after the trap is invoked is implementation dependent. In general, the user should assume that execution will eventually resume at the next instruction.

The various branch target registers and a subset of the GPRs will be in an undefined state after the execution of this instruction

6.5 Software Interrupt

swi



An interrupt/exception (SWI_0) is generated. It will be handled by the hardware similar to other internally generated exceptions; depending on the settings of various control registers, it is possible that the interrupt is ignored and no action is taken.

The various branch target registers and a subset of the GPRs will be in an undefined state after the execution of this instruction.

7 Privileged State

7.1 Overview

The architecture provides for user state and one or more privileged states. Each level of privilege grants access to different sets of resources and instructions. Some of these may be implementation specific.

It is intended that the user will access privileged resources by using the trap instruction as an API to request the operating-system or firmware to do certain operations. These can include operations such as atomic memory operations whose details can change from implementation to implementation. As such it is intended that the implementation of the trap instruction will, in at least some cases, be very low overhead.

This chapter describes the privileged state and instructions provided by a specific implementation of the architecture; other implementations will be different. In this implementation, there is 1 level of privileged state, controlled by the program status counter. The level is called hypervisor state. All state of the processor is accessible in hypervisor.

7.2 General state

7.2.1 Scratch Registers

There are a set of 8 32-bit scratch registers, written `$s0...$s7`, that are available in hypervisor state. It is intended that some of these will be reserved for keeping process-specific information, and others will be used for spilling user state and other interrupt processing.

7.2.2 Special Purpose Registers (SPR)

There are a variety of purpose-specific special purpose registers. Any writes to these registers are intended to be serializing.

SPR registers are accessible only in privileged modes (hypervisor) except FFT related SPR registers. Below is a summary of all special purpose registers in the GPTX DSP processor core.

The hypervisor privilege mode has Read/Write (RW) permissions on all SPRs listed below unless a Write Only (WO) or a Read Only (RO) is explicitly specified.

abbreviation	register name	permission	
PSC	Process Status and Control		0

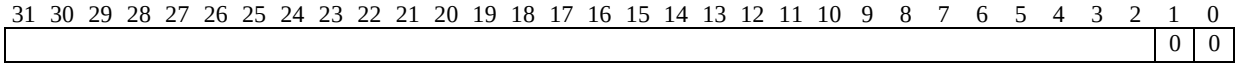
XPEN	External Pending		1
XCLR	External Pending Clear	WO	2
XSET	External Pending Set	WO	3
TCD0	Count Down Timer		4
TIMLO	Timer Base low		6
TIMHI	Timer Base High		7
CPUC	CPU Control		10
XEN	External Interrupt Enable		11
XLV	Exception Level Select		15
XMS	Exception Machine-check Select		16
TXB	Trap Base		18
HXB	Hypervisor Exception Base		1A
MXB	Machine-check Exception Base		1B
PSCH	Hypervisor Program Status and Control		1D
PSCM	Machine Check Program Status and Control		1E
XR0H	Level 0 Return Hypervisor		28
XS0H	Level 0 Status Hypervisor		29
IXA0H	Internal Exception Address Level 0 Hypervisor		2A
IXI0H	Internal Exception Instruction Level 0 Hypervisor		2B
XR1H	Level 1 Return Hypervisor		2C
XS1H	Level 1 Status Hypervisor		2D
IXA1H	Internal Exception Address Level 1 Hypervisor		2E
IXI1H	Internal Exception Instruction Level 1 Hypervisor		2F
XRM	Machine-check Return		30
XSM	Machine-check Status		31
IXAM	Internal Exception Instruction Machine-check		32
IXIM	Internal Exception Address Machine-check		33
XRT	Trap Return		34
XST	Trap Status		35
XRA	Exception Return Alternate		38
XSA	Exception Status Alternate		39
TIMCLO	Timer Base Status and Control Low		3E
TIMCHI	Timer Base Status and Control High		3F
DM0E	Data Address Match 0 Enable		40
DM1E	Data Address Match 1 Enable		41
DM0C	Data Address Match 0 Comapre		42
DM1C	Data Address Match 1 Comapre		43

DM0M	Data Address Match 0 Mask		44
DM1M	Data Address Match 1 Mask		45
DTCM	Data TCM Cmpare & Mask		46
ITCM	Instruction TCM Compare & Mask		47
SEG0	Segment 0 Compare & Mask		48
SEG1	Segment 1 Compare & Mask		49
SEG2	Segment 2 Compare & Mask		4A
SEG3	Segment 3 Compare & Mask		4B
SEG4	Segment 4 Compare & Mask		4C
SEG5	Segment 5 Compare & Mask		4D
SEGC0	Segment 0-3 Control Register		4E
SEGC1	Segment 4-5 Control Register		4F
TCD0C	Count Down Timer Control		5C
TRAPM	Trap Mask		5E
IM0E	Instruction Address Match 0 Enable		60
IM1E	Instruction Address Match 1 Enable		61
IM0C	Instruction Address Match 0 Compare		62
IM1C	Instruction Address Match 1 Compare		63
IM0M	Instruction Address Match 0 Mask		64
IM1M	Instruction Address Match 1 Mask		65
ICCTL	Instruction Cache Control	WO	68
SYNCCTL	Sync enable for each core	WO	71
PXIMASK	PXI Mask		72
PXITAG	PXI Tag		73
FFTW	FFT Weight	WO	78
FFTD	FFT Data		79
FFTC	FFT Control		7A
FRST	FFT software interrupt	WO	7B

7.2.2.1 Control

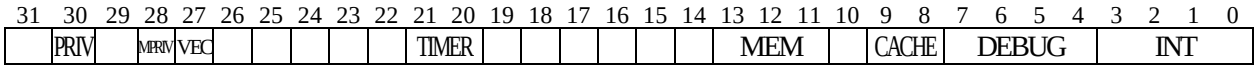
7.2.2.1.1 Program Counter (PC)

This 32-bit register contains the address of the instructions. It must be 4B aligned – i.e. lowest two bits are 0. It is used by the taddpci, taddpcil, aaddpci, aaddpcimb, and aaddpcilb instructions. It can be changed via the various branching instructions.



7.2.2.1.2 Process Status and Control Register (PSC)

The intent is that all information required to control a process will be stored in this register, so that only one register is needed to be saved and restored on interrupt. The fields in this register are:

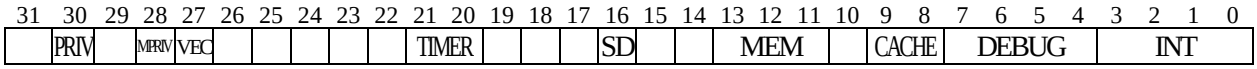


- PRIV: 1 bit, defines the privilege level of the current executing process
- USER (bit30): if 1, execute in user mode; if 0, execute in hypervisor mode.
- MEM_PRIV: 1 bit, defines the privilege level of the current memory access
- USER (bit28): if 1, access in user mode; if 0, access in hypervisor mode.
- VEC: 1 bit, vector unit control
- VD (bit27): vector disable.
- TIMER: 2 bits, timer control
- TM (bit20): timer increment/decrement enable.
- DE (bit21): disable timer on debug interrupt.
- MEM: 3 bits, memory control
- ITCM (bit11): instruction TCM enable.
- DTCM (bit 12): data TCM enable.
- NOSYNC (bit 13): sync enable.
- CACHE: 2 bits, cache enable
- IC (bit 8): instruction cache enable.
- DC (bit 9): data cache enable.
- DEBUG: 4-bit: controls various debug flags
- SS (bit 4): enable single step debug.
- BD (bit 5): enable debug on branch.
- IM (bit 6): enable instruction address matching.
- DM (bit 7): enable data address matching.
- INTERRUPT: 4-bit: controls whether various interrupts are recognized

- INTERRUPT_MC (bit 0): machine check interrupt enable.
- INTERRUPT_0 (bit 1): interrupt level 0 enable.
- INTERRUPT_1 (bit 2): interrupt level 1 enable.
- INTERRUPT_X (bit 3): external interrupt enable.

7.2.2.1.3 CPU Control Register (CPUC)

The CPU control register provides for CPU wide overrides of controls in the PSC. Conceptually, bits 0..31 of any value being written to the PSC are first AND-ed with bits in the CPUC before being written. The fields in this register is same as PSC register.

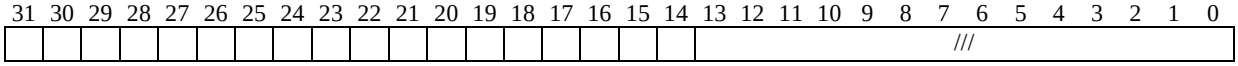


- SD: 1 bit, defines dispatch mode
- SD (bit16): dispatch single instruction per cycle.

7.2.2.2 Interrupt Controls

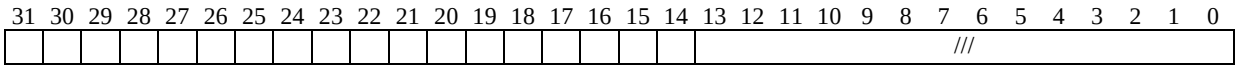
7.2.2.2.1 Machine-check exception base (MXB)

The internal exception base register is used to compute the address to which control is transferred when an internal interrupt/exception is taken and the interrupt is a machine-check interrupt. When an internal interrupt/exception is taken, control is transferred to the value of the MXB OR-ed with the exception number shifted by 8 bits.



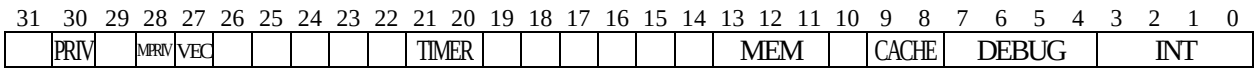
7.2.2.2.2 Hypervisor exception base (HXB)

The internal exception base register is used to compute the address to which control is transferred when an internal interrupt/exception is taken and the privilege level is raised to hypervisor. When an internal interrupt/exception is taken, control is transferred to the value of the HXB OR-ed with the exception number shifted by 8 bits.



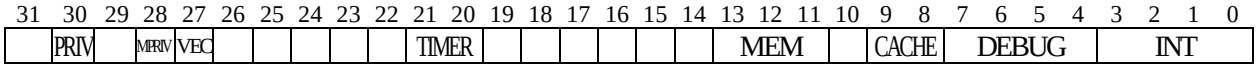
7.2.2.2.3 Process Status and Control Machine check (PSCM)

If an internal interrupt is a machine check, then the PSCM is copied into the PSC, becoming the new PSC; interrupts are appropriately disabled.



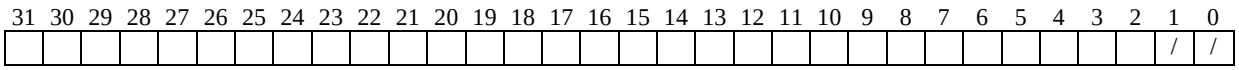
7.2.2.2.4 Process Status and Control Hypervisor (PSCH)

If an interrupt changes privilege level to hypervisor, then the PSCH is copied into the PSC, becoming the new PSC; interrupts are appropriately disabled.



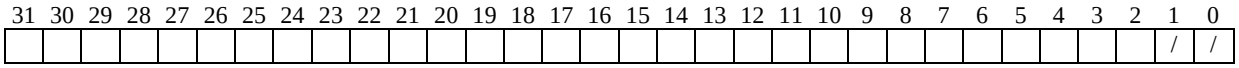
7.2.2.2.5 Exception Level Select Register (XLV)

Interrupt I is treated as a level 0/level 1 interrupt based on whether bit I of the XLV is 0 or 1.



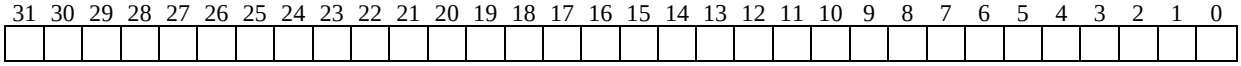
7.2.2.2.6 Exception Machine Check Select (XMS)

If interrupt I should be taken, but is suppressed because interrupts are disabled, and XMS bit I is 1, then it is treated as a machine check interrupt.



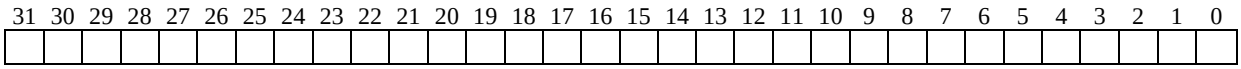
7.2.2.2.7 External Pending Register (XPEN)

The external interrupt pending register is used to signal that external interrupt I needs to be handled. Depending on the implementation, certain bits will persist until cleared by the interrupt handler using a write to the XPEN (or XCLR). On the current implementation, this is true for bits 0…23.



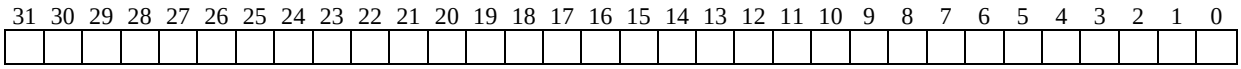
7.2.2.2.8 External Pending Clear on 1 Register (XCLR)

Writing a 1 to a bit in this write-only register will clear the corresponding bit of the XPEN register



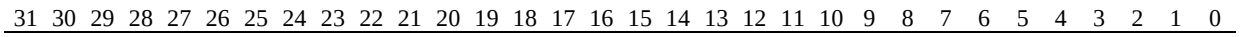
7.2.2.2.9 External Pending Set on 1 Register (XSET)

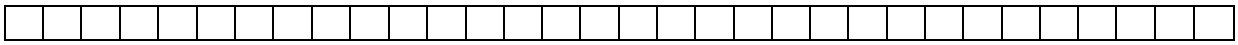
Writing a 1 to a bit in this write-only register will set the corresponding bit of the XPEN register



7.2.2.2.10 External Interrupt Enable (XEN)

Used to control external interrupts; if bit I of the XEN is 0, then bit I of XPEN is ignored.

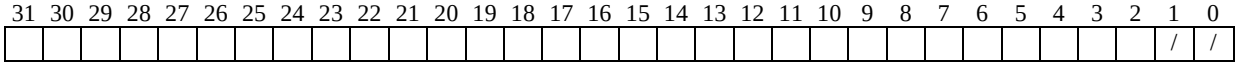




7.2.2.3 Interrupt Save/Restore

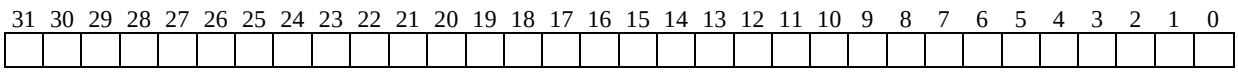
7.2.2.3.1 Level 0 Return Hypervisor (XR0H)

Return address for level 0 hypervisor mode interrupts. On a level 0 interrupt that switches to hypervisor privilege, the next instruction address that should be executed is stored in the XR0H



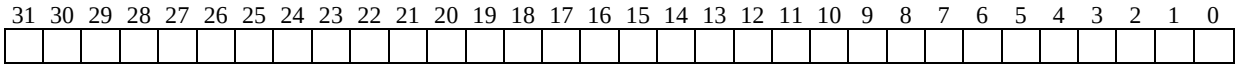
7.2.2.3.2 Level 0 Status Hypervisor (XS0H)

Saved PSC for level 0 hypervisor mode interrupts. On a level 0 interrupt that switches to hypervisor privilege, the value of the PSC at the time the interrupt was taken that should be executed is stored in the XS0H.



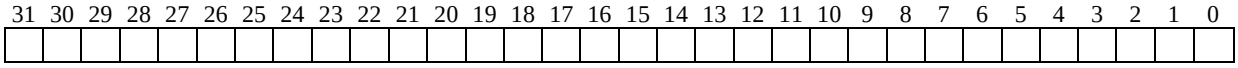
7.2.2.3.3 Internal Exception Instruction Level 0 Hypervisor (IXI0H)

Instruction causing the exception for level 0 hypervisor interrupts. On a level 0 interrupt that switches to hypervisor privilege, in certain situations the interrupt that caused the exception is copied to the 32 bits of the IXI0H.



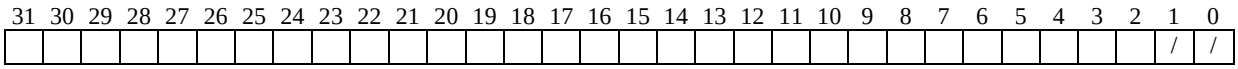
7.2.2.3.4 Internal Exception Address Level 0 Hypervisor (IXA0H)

Address causing the exception for level 0 hypervisor interrupts. On a level 0 interrupt that switches to hypervisor privilege, in certain situations the data address that caused the exception is copied to the of the IXA0H.



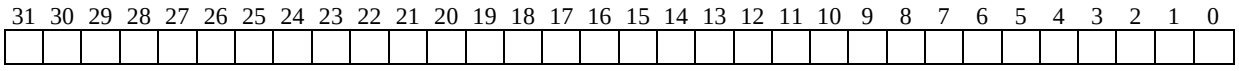
7.2.2.3.5 Level 1 Return Hypervisor (XR1H)

Return address for level 1 hypervisor mode interrupts. On a level 1 interrupt that switches to hypervisor privilege, the next instruction that should be executed is stored in the XR1H.



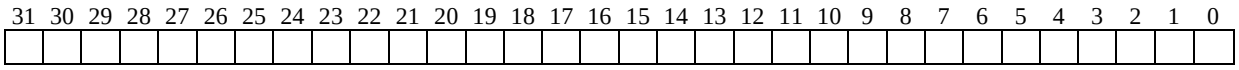
7.2.2.3.6 Level 1 Status Hypervisor (XS1H)

Saved PSC for level 1 hypervisor mode interrupts. On a level 1 interrupt that switches to hypervisor privilege, the value of the PSC at the time the interrupt was taken that should be executed is stored in the XS1H.



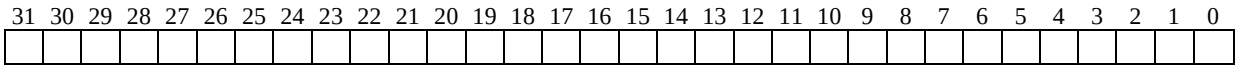
7.2.2.3.7 Internal Exception Instruction Level 1 Hypervisor (IXI1H)

Instruction causing the exception for level 1 hypervisor interrupts. On a level 1 interrupt that switches to hypervisor privilege, in certain situations the interrupt that caused the exception is copied to the32 bits of the IXI1H.



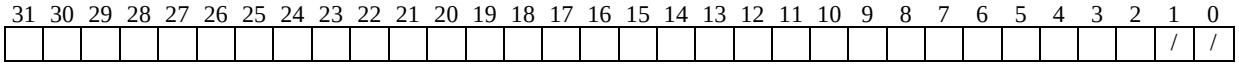
7.2.2.3.8 Internal Exception Address Level 1 Hypervisor (IXA1H)

Address causing the exception for level 1 hypervisor interrupts. On a level 1 interrupt that switches to hypervisor privilege, in certain situations the data address that caused the exception is copied to the of the IXA1H.



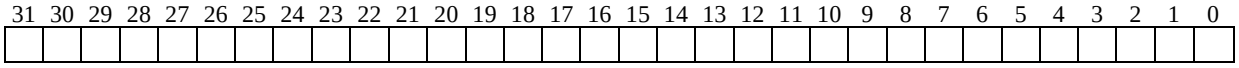
7.2.2.3.9 Machine Check Return (XRM)

Return address for machine check interrupts. On a machine check interrupt, the next instruction that should be executed after return form interrupt is stored in the XRM.



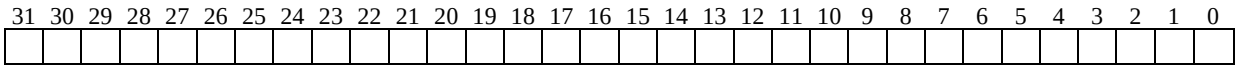
7.2.2.3.10 Machine Check Status (XSM)

Saved PSC for machine check interrupts. On a machine check interrupt, the value of the PSC at the time the interrupt was taken that should be executed is stored in the XSM.

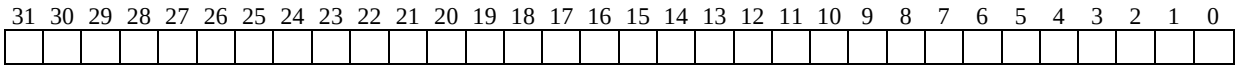


7.2.2.3.11 Internal Exception Instruction Machine Check (IXIM)

Instruction causing the exception for machine check interrupt. On a machine check interrupt, in certain situations the interrupt that caused the exception is copied to the 32 bits of the IXIM.

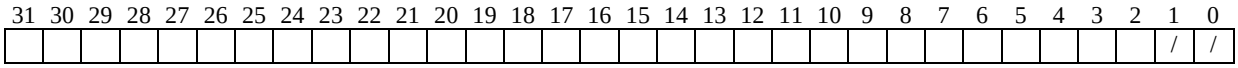


Address causing the exception for machine check. On a machine check interrupt, in certain situations the data address that caused the exception is copied to the IXAM.



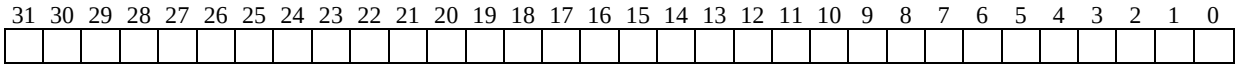
7.2.2.3.12 Exception Return Alternate (XRA)

Used to provide an alternate return address to the retfi instruction. On a retfi that use the XRA/XSA pair, this value is copied to the PC.



7.2.2.3.13 Exception Status Alternate (XSA)

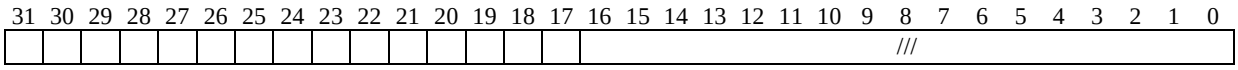
Saved PSC for alternate retfi instruction. On a retfi that use the XRA/XSA pair, this value is copied to the PSC register.



7.2.2.4 Trap

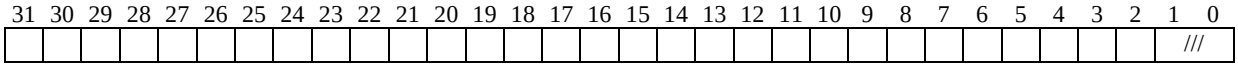
7.2.2.4.1 Trap Base Register (TXB)

The trap base register is used to compute the address to which control is transferred when an trap instruction is executed in user state. It must be aligned on a 2^{16}-byte boundary. When a trap is executed, control is transferred to the value of the TB OR-ed with the trap number shifted by 8 bits.



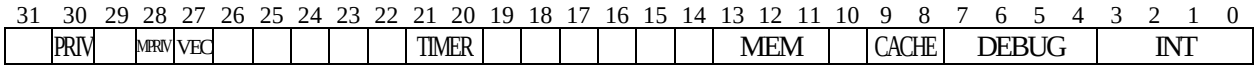
7.2.2.4.2 Trap Return Register (XRT)

The Trap Return register contains an instruction address. When a trap is executed, the XTR is set to the address of the next instruction after the trap. The XTR is copied to the PC when a return from trap instruction is executed. It is 4-byte aligned.



7.2.2.4.3 Trap Status Register (XST)

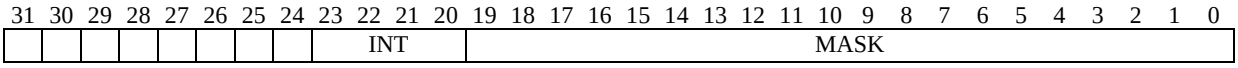
The value of internal exception status register is compatible with the format of the PSC. When a trap is executed, the IXS is set to the current value of the PSC. The TS is copied to the PSC when a return from trap instruction is executed.



7.2.2.4.4 Trap Mask (TRAPM)

The trap number in the trap instruction is and-ed with the 20-bit trap-mask register to obtain the actual trap number. This allows an implementation to restrict the number of traps it needs to implement.

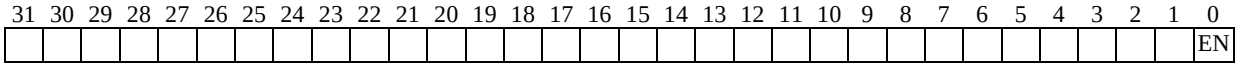
The 4 bits of the INT field are used to control the interrupt masking behavior when a trap is taken. If bits 20/21/22/23 are 1 then the interrupt mc/interrupt_0/interrupt_1/interrupt_x bits of the PSC respectively are cleared when a trap is executed.



7.2.2.5 Instruction Address Match

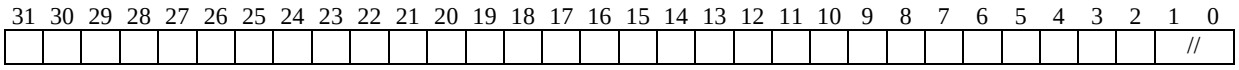
7.2.2.5.1 Instruction Address Match *N* Enable Register (IMnE)

Setting the low bit of the IMnE register enables the nth instruction address match control; n can have values of 0 or 1 in the current version.



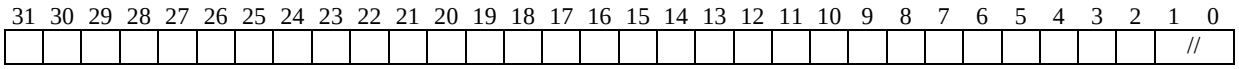
7.2.2.5.2 Instruction Address Match *N* Mask Register (IMnM)

The value in the IMnM register is and-ed with the instruction address prior to comparison. It is 4 byte aligned.



7.2.2.5.3 Instruction Address Match *N* Compare Register (IMnC)

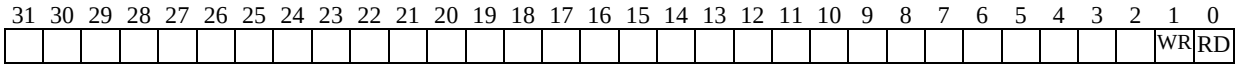
The value in the IMnC register is compared with result of and-ing with IMnM. It is 4 byte aligned.



7.2.2.6 Data Address Match

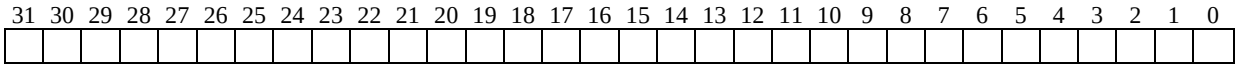
7.2.2.6.1 Data Address Match *N* Enable Register (DMnE)

Bit 0 of the DMnE register enables the nth data address match for loads. Bit 1 of the DME enables the nth data address match for stores.



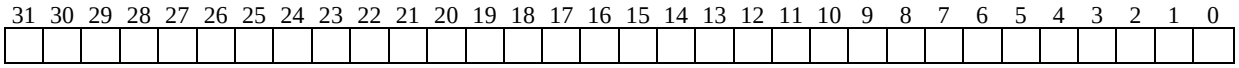
7.2.2.6.2 Data Address Match *N* Mask Register (DMnM)

The value in the DMnM register is and-ed with the data address prior to comparison.



7.2.2.6.3 Data Address Match *N* Compare Register (DMnC)

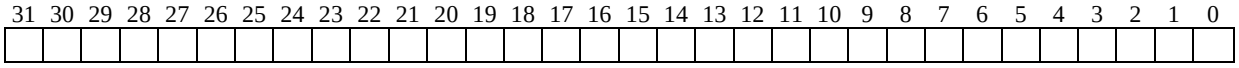
The value in the DMnC register is compared with result of and-ing with DMnM

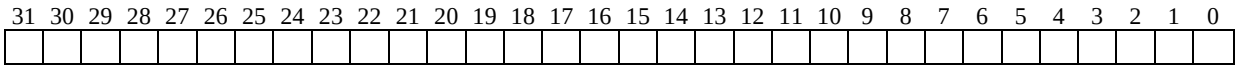


7.2.2.7 Timer

7.2.2.7.1 Time Base (TIMHI. TIMLO)

The 64-bit timer register (TIMHI+TIMLO), that is incremented at some frequency



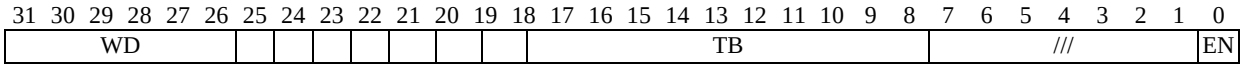


The Time base register will be incremented once every 64 clocks when:

- The EN bit in the TIMCHI and TIMCLO is 1 and
- The TM bit in the CPUC is 1 and
- The TM bit in the PSC is 1

7.2.2.7.2 Time Base Status and Control (TIMCHI, TIMCLO)

Enables increment. Enables interrupts and exceptions, records status of pending exceptions.

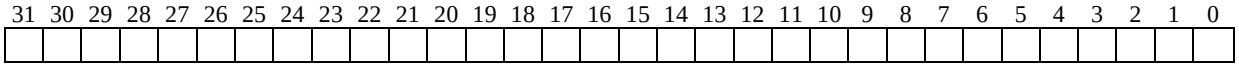


The Time Base Control SPR (TIMCHI, TIMCLO) has the following fields:

- EN: 1-bit: Increment Enable: when 1, incrementing is enabled.
- TB: 10-bit: Time Base Exception Enable: an exception will be generated when a bit in the TB is 1 and the corresponding bit (8…17) in the TIM generates a carry-out
- WD: 6-bit: Watchdog Exception Enable: an exception will be generated when a bit in the WD is 1 and the corresponding bit (26..31) in the TIM generates a carry out

7.2.2.7.3 Count Down Timer *N* (TCD*n*)

The 32-bit count-down registers. They can generate a timer interrupt when counting down from 1 to 0.

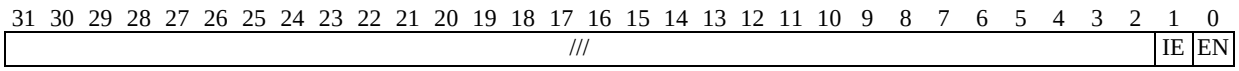


The Count-down timer SPR (TCDn) will be decremented once every cycle when:

- The EN bit in the TCDnC is 1 and
- The DM bit in the CPUC is 1 and
- The DM bit in the PSC is 1

7.2.2.7.4 Count Down Timer *N* Control (TCD*n*C)

Controls whether the n^{th} count down timer will decrement. Also enables/records interrupts.



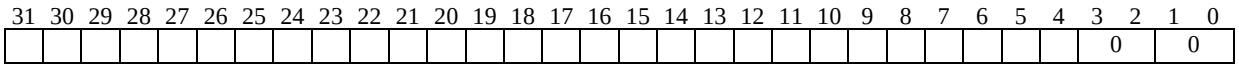
The Count-down timer control (TCDnC) has the following fields:

- EN: 1 bit: Count Down Timer Decrement Enable: when 1, count-down timer decrementing is enabled.
- IE: 1 bit: Count Down Interrupt Enable: when 1, count-down timer exceptions are enabled; when the counter goes from 1 to 0, a count-down timer exception is generated.

7.2.2.8 Cache Control

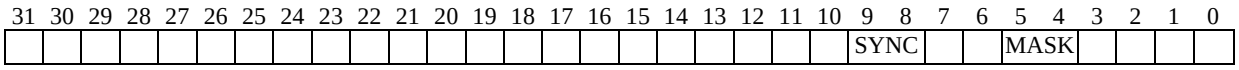
7.2.2.8.1 Instruction Cache Control (ICCTL)

Writing an address to the ICCTL register causes all 4 ways of the set to be invalidated.



7.2.2.9 Sync Control

7.2.2.9.1 Sync Control (SYNCCTL)



The sync control register is used to control the configuration of cache coherency. It can be used:

When a sync command is written, each cores participation in the multi-core synchronization can be enabled or disabled. The 2 bits of the MASK and SYNC registers correspond to each of the two cores. If the MASK bit for the core is 0, its synchronization status is left unchanged; otherwise if the SYNC bit for that core is 1, it participates in synchronization, otherwise it does not.

7.2.2.10 Segment configuration

7.2.2.10.1 Segment Control 0 (SEGC0)

Segment control. There are 6 segments, and 6 SEGxC field, every field configure the segment hypervisor mode read/write, user mode read/write, and cacheability.

31 30 29 28 27 26 25 24	23 22 21 20 19 18 17 16	15 14 13 12 11 10 9 8	7 6 5 4 3 2 1 0
SEG3C	SEG2C	SEG1C	SEG0C

- SEG3C: 7-bit, segment 3 control
- NOSYNC (bit 30): not sync
- HX (bit 30): hypervisor mode execution permission
- HR (bit 29): hypervisor mode read permission
- HW(bit 28): hypervisor mode write permission
- UX (bit 27): user mode execution permission
- UR (bit 26): user mode read permission
- UW(bit25): user mode write permission
- CACHE(bit 24): if 1 is cached address; if 0 is uncached address
- SEG2C: 7-bit, segment 2 control
- NOSYNC (bit 23): not sync
- HX (bit 22): hypervisor mode execution permission
- HR (bit 21): hypervisor mode read permission
- HW(bit 20): hypervisor mode write permission
- UX (bit 19): user mode execution permission
- UR (bit 18): user mode read permission
- UW(bit 17): user mode write permission
- CACHE(bit 16): if 1 is cached address; if 0 is uncached address
- SEG1C: 7-bit, segment 1 control
- NOSYNC (bit 15): not sync
- HX (bit 14): hypervisor mode execution permission
- HR (bit 13): hypervisor mode read permission
- HW(bit 12): hypervisor mode write permission
- UX (bit 11): user mode execution permission
- UR (bit 10): user mode read permission
- UW(bit 9): user mode write permission
- CACHE(bit 8): if 1 is cached address; if 0 is uncached address

- SEG0C: 7-bit, segment 0 control
- NOSYNC (bit 7): not sync
- HX (bit 6): hypervisor mode execution permission
- HR (bit 5): hypervisor mode read permission
- HW(bit 4): hypervisor mode write permission
- UX (bit 3): user mode execution permission
- UR (bit 2): user mode read permission
- UW(bit 1): user mode write permission
- CACHE(bit 0): if 1 is cached address; if 0 is uncached address

7.2.2.10.2 Segment Control 1 (SEGC1)

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16	15 14 13 12 11 10 9 8	7 6 5 4 3 2 1 0
///	SEG5C	SEG4C

- SEG5C: 7-bit, segment 5 control
- NOSYNC (bit 15): not sync
- HX (bit 14): hypervisor mode execution permission
- HR (bit 13): hypervisor mode read permission
- HW(bit 12): hypervisor mode write permission
- UX (bit 11): user mode execution permission
- UR (bit 10): user mode read permission
- UW(bit 9): user mode write permission
- CACHE(bit 8): if 1 is cached address; if 0 is uncached address
- SEG4C: 7-bit, segment 4 control
- NOSYNC (bit 7): not sync
- HX (bit 6): hypervisor mode execution permission
- HR (bit 5): hypervisor mode read permission
- HW(bit 4): hypervisor mode write permission
- UX (bit 3): user mode execution permission
- UR (bit 2): user mode read permission
- UW(bit 1): user mode write permission
- CACHE(bit 0): if 1 is cached address; if 0 is uncached address

7.2.2.10.3 Segment 0 (SEG0)

Segment 0 base address and size. The base address is 64KB aligned, and the size unit is 64KB.

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

7.2.2.10.4 Segment 1 (SEG1)

Segment 1 base address and size. The base address is 64KB aligned, and the size unit is 64KB.

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

7.2.2.10.5 Segment 2 (SEG2)

Segment 2 base address and size. The base address is 64KB aligned, and the size unit is 64KB.

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

7.2.2.10.6 Segment 0 (SEG3)

Segment 3 base address and size. The base address is 64KB aligned, and the size unit is 64KB.

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

7.2.2.10.7 Segment 4 (SEG4)

Segment 4 base address and size. The base address is 64KB aligned, and the size unit is 64KB.

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

7.2.2.10.8 Segment 5 (SEG5)

Segment 5 base address and size. The base address is 64KB aligned, and the size unit is 64KB.

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

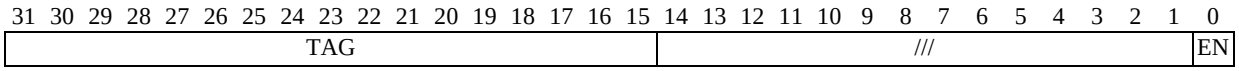
7.2.2.11 Private eXternal Interface (PXI)

If a load/store real address satisfies the following formula, then the read/write request will be sent to the PXI request queue:

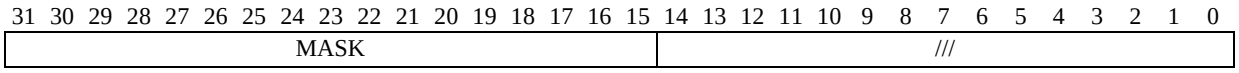
```
if pxitag[0] = 1 and
  ( address[31:15] and not pximask[31:15] ) = pxitag[31:15] then
  pxi_request = 1;
else
  pxi_request = 0;
end if
```

7.2.2.11.1 PXI Tag (PXITAG)

When the core is reset, the PXIMASK enable bit (bit 0) is cleared, so the chip comes up with PXI disabled.



7.2.2.11.2 PXI Mask (PXIMASK)



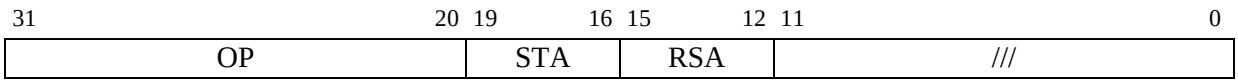
7.2.2.12 FFT control

Definding…

7.2.3 Instructions

7.2.3.1 Copy Scratch from GPR

ssetr $sta,$rsa

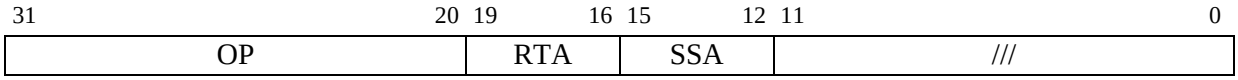


STA ← [RSA]

Copy the value in general-purpose register RSA to scratch register STA.

7.2.3.2 Copy GPR from Scratch

rsets $rta,$ssa

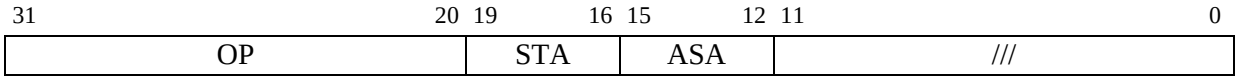


RTA ← [SSA]

Copy the value in scratch register SSA to general-purpose register RTA.

7.2.3.3 Copy Scratch from Address

sseta $sta,$asa

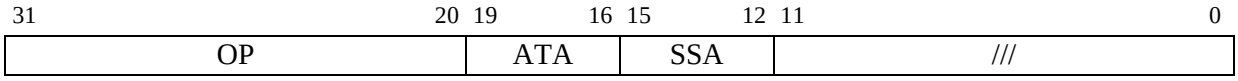


STA ← [ASA]

Copy the value in address register ASA to scratch register STA.

7.2.3.4 Copy Address from Scratch

asets $ata,$ssa

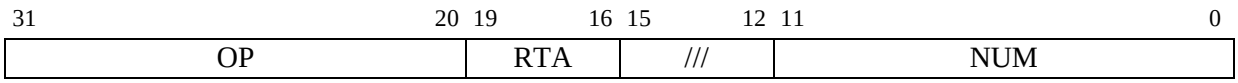


```
ATA ← [SSA]
```

Copy the value in scratch register SSA to address register ASA.

7.2.3.5 Copy GPR from Special Purpose Register

`rsetspr $rta,num`

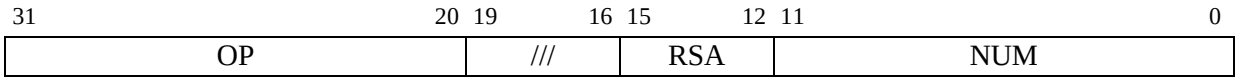



```
RTA ← SPR( NUM )
```

Copy the value in the special purpose register identified by NUM to general-purpose register RTA. If there are multiple values of NUM that are used by the `sprsetr` instruction to modify the same physical register, and at least one of them results in a copy, then that is the value that should be used by the `rsetspr` register.

7.2.3.6 Copy Special Purpose from GPR

`sprsetr $rsa,num`




```
SPR( NUM ) ← [RSA]
```

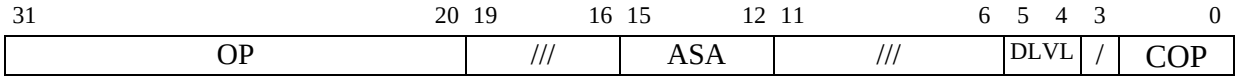
Copy the value in general-purpose register RSA to special purpose register identified by NUM. Depending on the state being altered, it may need to be preceded/succeeded with a barrier or fence. It is possible that, depending on the register being addressed, the value in RSA is not copied, but is used to modify the existing value. The possible actions are:

- Copy
- Set on one (i.e. or with RSA value)
- Clear on one (i.e. and with complement of RSA value)
- Flip on one (i.e. xor with the RSA value)

It is possible that multiple of these behaviors may be desired on the same register; in that case, we will use different NUM values to address the same physical SPR, but perform different actions on it.

7.2.3.7 Data Cache Control

dctl_cop_dlvl $asa



This instruction affects the data cache line(s) ASA. The COP field specifies the operations that are to be performed. The exact combination of operations/levels available are implementation dependent.

The DLVL field specifies how many levels of the cache hierarchy are affected. But DLVL is not supply in this edition, default DLVL field is 0.

Currently, the following values are defined for COP:

- COP=0: invalidate all ways of the set that would contain ASA
- COP=1: flush an entry with the way identified by the lower bits of ASA in the set that would contain ASA, and mark it as invalid
- COP=2: unlock all ways of the set that would have contained ASA
- COP=3: unlock the cache line containing ASA
- COP=5: if ASA is not in the cache, fetch it; then lock the cache line containing ASA
- COP=6: if ASA is in the cache, flush it, and mark the as invalid

The assembler uses the mnemonics `dctl_invalid` / `dctl_flush` / `dctl_open` / `dctl_unlock` / `dctl_lock` / `dctl_sync` / `dctl_prefetch` for COP=0/1/2/3/5/6.

For the current chip, the following combinations are valid:

- `dctl_invalid`
- `dctl_flush` (the lower 2 bits of ASA will be used to identify one of four ways)
- `dctl_open`
- `dctl_unlock`
- `dctl_lock`
- `dctl_sync`

A dctl_invalid_l1 $a0 finds the row that the address in $a0 will occupy in the L1, and invalidate all 4 lines in that L1 row

A dctl_flush_l1 $a0 finds the row that the address in $a0 will occupy in the L1, and use the lower 2 bits to identify one of the 4 ways. It will then flush that row/way. If the line is invalid, then it is a nop. If the line is valid, but not dirty, then it is invalidated. If the line is valid and dirty, it is castout to the next level (L2 or memory) and marked as invalid.

A dctl_sync_l1 $a0 finds looks for a line containing the address $a0 in L1. If the line is not present in L1, then it is a nop. If the line is valid, but not dirty, then it is invalidated. If the line is valid and dirty, it is castout to the next level (L2 or memory) and marked as invalid.

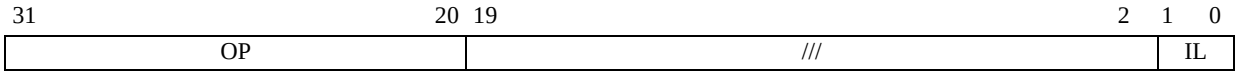
A dctl_lock_l1 $a0 locks the line containing $a0 into the L1. If the line is not present, it is fetched and then locked. If the line is already locked, this is a nop. It expects that the address is 32B aligned, and that it has read and write permissions to the line.

A dctl_unlock_l1 $a0 unlocks the line containing $a0. If the line is not locked, this is a nop.

A dctl_open_l1 $a0 unlocks all lines in the row that the address in $a0 would occupy.

7.2.3.8 Idle

idle_il




```
Halt execution
```

Execution of the idle instruction halts execution. Execution will resume on an interrupt. At the interrupt, the relevant interrupt return register will hold the address of the idle instruction.

The IL field is a hint for an external power controller.

7.3 Trap

The trap instructions will:

- Save address of the trap + 4 in XRT
- Save the PSC in XST
- Clear the INTERRUPT_MC, INTERRUPT_0, INTERRUPT_1, INTERRUPT_X bits of the PSC if bits 20/21/22/23 respectively of TRAPM are set.
- Clear the PRIV_USER bits of the PSC, run in hypervisor mode
- Prevent any instructions from the fall-through path from being pre-fetched.
- "And" the trap number specified in the trap register with the TRAPM register, to obtain a trap number
- Branch to TXB + 256 * `trap_number`.

7.4 Debug

7.4.1 Overview

There are 4 debug mechanisms available:

- Single step: an exception will occur after the execution of every instruction
- Branch trace: an exception will occur after the execution of a branch instruction (whether it is taken or not)
- Instruction address compare: an exception will occur before the execution of an instruction that satisfies one of the provided address patterns.
- Data address compare: an exception will occur before the execution of a memory operation that satisfies one of the provided address patterns.

These exceptions are enabled by bits in PSC.

7.4.2 Single Step

The processor will take a single-step interrupt after the execution of any instruction. After the instruction is executed:

- The address of the instruction that was just executed is stored in an IXA SPR
- The instruction is copied into an IXI SPR
- The address of the next instruction to be executed is stored in the XR SPR.
- The current PSC is saved in a XS SPR
- The interrupt level(s) associated with the single-step interrupt are cleared
- The interrupt handler address is determined by combining the appropriate interrupt base with the interrupt number for the single step debug, and execution branches to that address.

Single-step debug requires that:

- the DEBUG_SS bit in the CPUC and PSC is set
- the appropriate interrupt bit is enabled in the PSC

7.4.3 Branch debug

The processor can take a branch debug interrupt after the execution of any branch instruction. After the branch is executed:

- The address of the branch instruction that was just executed is stored in an IXA SPR
- The branch instruction is copied into an IXI SPR
- The address of the next instruction (i.e. the target if taken, otherwise the fall-through address) to be executed is stored in the XR SPR.
- The current PSC is saved in a XS SPR
- The interrupt level(s) associated with the branch debug interrupt are cleared
- The interrupt handler address is determined by combining the appropriate interrupt base with the interrupt number for the branch debug, and execution branches to that address.

Branch debug requires that:

- the DEBUG_BD bit in the CPUC and PSC is set
- the appropriate interrupt bit is enabled in the PSC

7.4.4 Instruction address match

There are 2 sets of instruction address match registers available. Each set consists of an enable, a mask and a compare, called the `IMnE`/ `IMnM`/ `IMnC`, where n=0 or n=1

An instruction address match occurs if, for any instruction match register set:

- `IMnE EN` is 1
- The instruction address and-ed with `IMnM` is equal to `IMnC`.

If an instruction address match occurs:

- The instruction is not executed
- The address of the instruction that was matched is stored in an XR SPR
- The instruction is copied into an IXI SPR
- The current PSC is saved in a XS SPR
- The interrupt level(s) associated with the instruction match interrupt are cleared
- The interrupt handler address is determined by combining the appropriate interrupt base with the interrupt number for the instruction match debug, and execution branches to that address.

Instruction address match requires that:

- the DEBUG_IM bit in the CPUC and PSC is set
- the appropriate interrupt bit is enabled in the PSC

7.4.5 Data address match

There are 2 sets of data address match registers available. Each set consists of an enable, a mask and a compare, called the `DMnE`/ `DMnM`/ `DMnC`, where n=0 or n=1.

A data address match occurs if, for any data address match register set:

- A load instruction is being executed, and `DMnE LD` bit is 1 or a store instruction is being executed and `DMnE` ST bit is 1
- The data address and-ed with `DMnM` is equal to `DMnC`.

If a data address match occurs:

- The instruction is not executed
- The instruction address of the load/store that was to be executed is stored in an XR SPR
- The data address that matched is stored into an IXA
- The instruction is copied into an IXI SPR
- The current PSC is saved in a XS SPR
- The interrupt level(s) associated with the data address match interrupt are cleared
- The interrupt handler address is determined by combining the appropriate interrupt base with the interrupt number for the data address match debug, and execution branches to that address.

Data address match requires that:

- the DEBUG_DM bit in the CPUC and PSC is set
- the appropriate interrupt bit is enabled in the PSC

7.5 Timers

7.5.1 Overview

The timer consists of a 32-bit time-base register (TIM) that is incremented at the clock-frequency divided by 64, and a 32-bit count-down registers (TCD0) that are decremented at the frequency.

When the TCD0 counts down to 0, or when increment of the TIM generates a carry-out from certain bit positions, it is possible to cause a timer exception. These are controlled and recorded by the time-base control and status register (TIMC) and the count-down control and status registers (TCD0C)

7.5.2 Exceptions

There are three exception types from the timer subsystem:

- Count Down Interrupt: when TCD0 counts down to 0 and the IE bit in the TCD0C register is set
- Time Base Interrupt: when there is a carry out from some subset of bits 8 through 17 of the TIM. These bits are selected using the TB bits of the TIMC.
- Watchdog Interrupt: when there is a carry out from some subset of bits 26 through 31 of TIM. These bits are selected using the WD bits of the TIMC

These exceptions are treated as external interrupts by the core. They set bits in the external exception pending register (XPEN), and will be processed as other external exceptions.

7.6 Interrupts & Exceptions

7.6.1 Overview

Interrupts are deviations from the normal code flow caused by events external to the instruction sequence being executed, though possibly triggered by a behavior initiated by some prior instruction. They are asynchronous.

Exceptions are deviations from the normal code flow related to the execution of an instruction, and are synchronous.

Interrupts/exceptions are split into internal interrupts/exceptions that are typically generated from within a core, and external interrupts, that are generated from outside the core.

An interrupt can be treated as a machine check, a level-0 or level-1 interrupt.

When an interrupt is encountered, various information is saved:

- The address of the next instruction to be executed is saved to an interrupt return register
- The current status of the PSC is saved to an interrupt status register
- The instruction that caused the interrupt may be written to an interrupt instruction register
- If the interrupt was caused by an address, such as a data address un-align, the offending address is written to an interrupt address register.

Then, an interrupt base register is selected based on the interrupt level and the final privilege level. Control is transferred to the address formed by or-ing this base with the interrupt number * 256.

A new value is loaded into the PSC. This is an appropriately modified copy of the existing PSC or a copy from a PSCH/PSCM SPRs. The interrupt for the interrupt level being taken, and all lower

interrupts are disabled. This is made visible by clearing the corresponding enable bits in the PSC. When interrupt processing is complete, a retfi instruction is used to restore the PSC and PC from the saved values

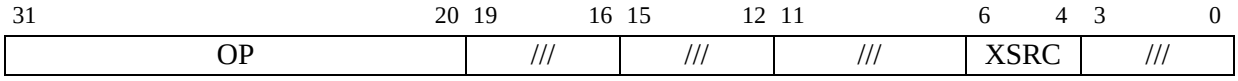
7.6.2 Registers

- PSC (enables): controls whether an interrupt can actually interrupt program execution.
- Base address: There are three of these registers, for trap, hypervisor and machine check interrupts that identifies the location at which exception processing should proceed. These are the TXB, HXB and MXB. The interrupt or trap number *256 is or-ed with these registers to form the interrupt handler address.
- External Pending: The external pending exception SPR (XPEN) holds bits for external interrupts. Some bits are “sticky” and others are “transparent”. Transparent bits can only be driven by external interrupt lines. The sticky bits can also be written directly by writing to this SPR, in which case the processor will act as though the bits were written by an external exception. Sticky bits can be set or cleared in the XPEN by writing a 1 at that bit position to the XSET/XCLR SPRs
- External Enable: The enable external exception SPR (XEN) turns on and off external exceptions; if bit I of the XEN is clear, a 1 in the position I of the XPEN does not cause an interrupt
- Interrupt level: The exception level select SPR (XLV) determines whether interrupt I should be a level-0 (bit I=0) or level-1 interrupt (bit I=1)
- Interrupt/Exception return: There are multiple of these registers, to enable interrupts to be nested. They store the address of the next instruction that should be executed. They are the XRT, XRA, XR0H, XR1H, XRM.
- Interrupt/Exception status: There are multiple of these registers, one for each return. They store the PSC at the time the interrupt was taken. They are XST, XSA, XS0H, XS1H, XRSM
- Exception instruction: Some internal exceptions will save the instruction that caused the exception in an exception register. This enables the code to disassemble the instruction without having to load it from memory. There are separate registers for different interrupts, and include IXI0H, IXI1H, IXIM. The instructions are saved into these registers.
- Exception address: For some address related internal exceptions, such as alignment, IXA SPR will store the address that caused the exception. In the case of the single step and branch trace exceptions, IXA SPR will contain the address of the instruction that was executed[5]. The registers are IXA0H, IXA1H, IXAM

[5] After a single step/branch trace debug, the IXRSPR contains the address of the instruction to resume execution, and the IXASPR contains the address of the instruction that was just executed.

7.6.3 Return from Interrupt instruction

retfi_xsrc




```
if( XSRC = 000b )
  PC ← XRT
  PSC ← XST
else if( XSRC = 001b )
  PC ← XRM
  PSC ← XSM
else if( XSRC = 100b )
  PC ← XR0H
  PSC ← XS0H
else if( XSRC = 101b )
  PC ← XR1H
  PSC ← XS1H
 else if( XSRC = 110b )
  PC ← XRA
  PSC ← XSA
```

Resume execution at the address stored in the appropriate interrupt return target SPR, and simultaneously set the PSC using the value stored in the interrupt return status SPR.

In the assembler, the mnemonics `retfi_trap/ retfi_mc/ retfi_l0h / retfi_l1h / retfi_alt` are used for XSRC=0/1/4/5/6.

7.6.4 Interrupts

7.6.4.1 Machine Check

The Machine-check interrupt is intended to force the CPU to halt processing immediately, and rest. It is to be used only when a catastrophic failure has occurred.

7.6.4.2 External Interrupt

There are assorted asynchronous interrupts that are generally intended to allow handling of external device events. The number and manner in which they are invoked is implementation specific.

Each of these will write a 1 to the XPEN register. When XPEN bit I is 1, and XEN is 1, and the corresponding interrupt (based on XLVL/XMS) is enabled, interrupt I is taken. This is gated by the INTERRUPT_X bit of the PSC & CPUC; if either is 0 the interrupt is not taken.

7.6.5 Internal Exceptions

7.6.5.1 Protection

Protection exceptions are raised when:

- A program attempts to execute an instruction from a segment for which it does not have execute permission at its current privilege level.
- A program attempts to read data from a segment for which it does not have read permission at its current privilege level.
- A program attempts to write data to a segment for which it does not have write permission at its current privilege level.

7.6.5.2 Alignment

Alignment exceptions occur when:

- A branch target register is loaded with a value that is not a multiple of 4.
- A load or store is attempted at an address that is not a multiple of the size of the value being accessed.

7.6.5.3 Instruction Format

Instruction format exceptions occur when:

- There is no instruction with that opcode
- There is an instruction with that opcode, but the current privilege level does not permit its execution
- There is a field with an invalid value
- The current implementation does not support the instruction in hardware

7.6.5.4 Debug

Depending on the debug facilities available, an exception will be raised to allow for the program to be debugged. This can happen:

- Every cycle: if single step mode is enabled
- Every branch: if branch mode is enabled
- Instruction address match: if instruction breakpoint registers are enabled, and the address of the instruction to be executed matches the address in one of the registers
- Write address match: if data breakpoint registers are enabled, and the address of a store matches the address in one of the data registers, and that data breakpoint register is watching for writes.
- Reads address match: if data breakpoint registers are enabled, and the address of a load matches the address in one of the data registers, and that data breakpoint register is watching for reads.

7.6.5.5 Timer

A timer exception will trigger when the count-down timer reaches 0, or one of the enabled time base bits generate a carry out.

7.6.5.6 List of interrupts

The following is a list of internally generated asynchronous interrupts and synchronous exceptions. They are numbered starting at 0.

0	MACHINE_CHECK	Machine-check
1	TIMER_BASE	Timer base exception
2	TIMER_WATCH	Timer watchdog exception
3	TIMER_COUNT0	Timer countdown exception from timer 0
4	UNAVAIL_VEC	Vector unit not available
5		
6	INSN_BAD	Undefined instruction
7	INT_DIVIDE	An integer divide by zero
8	INSN_PERM	Attempted to execute an instruction without appropriate permission
9		
10		
11	INSN NOEXEC	Attempting to execute an instruction from a no-exec page
12	DATA_ALIGN	Attempted to load/store a value from/to a non-aligned value
13	DATA_CRSD	Attempted to load/store a value from a cross domain address.
14	DATA_NOREAD	Attempting to load a value from a no-read page
15	DATA_NOWRITE	Attempting to store a value from a no-write page
16		
17		
18		
19		
20		
21	DBG_STEP	Debug single step exception
22	DBG_BRANCH	Debug branch exception
23	DBG_INSN	Debug instruction match
24	DBG_DATA	Debug data match
25	SWI_0	Software Interrupt
26	SGI_0	Software Interrupt
27	SGI_1	Software Interrupt
28	MPINT_FIQ	Fast Interrupt Request
29	MPINT_IRQ	Interrupt Request
30	FFT INTERRUPT	FFT calculation completed
31		
32		
33		
34		
35		

7.6.5.7 Interrupt priority

If an instruction can take multiple interrupt, they shall be taken in the following priority order

- INSN_BAD
- INSN_PERM
- INT_DIVIDE
- VEC_TYPE
- VEC_LENGTH
- DATA_ALIGN
- DBG_DATA
- DBG_INSN
- SWI_0
- DBG_STEP
- DBG_BRANCH

8 Interrupt Subsystem

8.1 Overview

This chapter describes the organization of the interrupt sub-system, including the special-purpose registers associated with it.

Another motivation for the organization of the interrupt sub-system as described in this chapter is to allow for increased flexibility and control of the interrupts. This allows a core implementing this architecture to be combined with different external exceptions.

8.2 Interrupts

8.2.1 Interrupt Number

The interrupts are assumed to be numbered 0 to N-1, assuming N interrupts are supported. It is assumed that at most 32 interrupts are supported. Of these, 0 is reserved for the machine-check interrupt. The interrupt priority is set up so that, within an interrupt level, the lowest numbered interrupt has the highest priority.

8.2.2 Interrupt Levels

There are 4 interrupt levels:

- Trap
- Level 1
- Level 0
- Machine Check

Each of these levels has a pair of associated registers:

- Interrupt Return Address: the address at which instruction processing should resume when the interrupt has been processed
- PSC Save: the value of the PSC at the time the interrupt is taken, and thus the value that should be restored after processing the interrupt.

We distinguish between Level 0 & 1 interrupts that are taken into hypervisor mode and give different pairs of registers. These registers are known as:

- XRT/XST: Trap
- XR1H/XS1H: Level 1 Hypervisor
- XR0H/XS0H: Level 0 Hypervisor
- XRM/XSM: Machine-check

If there are multiple levels of interrupts pending then the priority is machine-check, level-0, level-1, and finally trap.

8.2.3 Interrupt Base Registers

The destination of a trap is computed by adding a fixed offset based on the interrupt number (for interrupts) or trap number (for trap instructions) to a base register. The value in the base register is expected to be page aligned. The offset for each instruction is 256B (i.e. 64 instructions).

The base register for traps is the TXB. Thus, for trap N, the new program counter is:

PC ← TXB + 256*N

The base register for machine-check interrupts is the MXB. For interrupt I, the new program counter is:

PC ← MXB + 256*I

The base register for interrupts into hypervisor mode is the HXB. For interrupt I, the new program counter is:

PC ← HXB + 256*I

8.2.4 Interrupt Level Select

There is an exception level select register, XLV, responsible for determining whether an interrupt is level 0 or level 1. There is one bit for each interrupt. If bit I is clear, interrupt I is treated as a level 0 interrupt, otherwise as a level 1 interrupt.

Note that interrupt 0 is the machine check interrupt, and is always treated as a machine check interrupt, independent of the value of XLV_0.

8.2.5 Final PSC

When an interrupt is taken, the PSC value is changed:

- The value of the PSCH are copied to the PSC
- The interrupt enable bit(s) of the PSC are cleared, based on the interrupt being taken.
- Level 1: interrupt_1 is cleared
- Level 0, 1: interrupt_0, interrupt_1 are cleared
- Machine Check: interrupt_mc, interrupt_1, interrupt_0 are cleared

If the interrupt is a machine-check interrupt, then the PSC is initialized to a copy of the value stored in the PSCM register, and the interrupt enable bits are cleared as above.

8.2.6 Machine Check

The machine check exception will be generated if:

- The processor detects an internal fault
- The program sets bit 0 of the XPEN with XEN bit 0 set
- The program would have taken a level_0/ level_1 interrupt number I, but interrupt_0/ interrupt_1 in the PSC is disabled, and XMS register bit I is set.

This last case is designed to ensure that if a critical interrupt handler, the processor can report the program and go into some kind of debug or fail-safe state.

Note that the interrupt number used is bit I

8.2.7 External Interrupt Support

There are several registers that are designed primarily to support external interrupts, and other interrupts that can be deferred. These registers are the exception enable (XEN) and exception pending (XPEN) registers. Both of these are per-interrupt registers, i.e. with one bit for each interrupt.

When bit I of the XEN is set, interrupts are masked off, and will not happen. When bit I is clear, interrupts are enabled. If an interrupt I is reported, then (depending on the interrupt type) the implementation may choose to set the corresponding bit in the interrupt pending register. Some these bits may be “sticky” – i.e. after the interrupt is reported the bit will persist in the XPEN until the bit is cleared. Alternatively, the bit may be “transparent” – i.e. the bit is high only as long as the external line driving the bit is high.

Clearing and setting individual sticky bits in the XPEN register can be accomplished by writing to the XCLR and the XSET registers. These registers will and and-complement/or the bits of the XPEN with the written bits. Thus, they are clear-on-1/set-on-1.

8.2.8 Interrupt Exception Support

When an internal exception occurs, the processor will save information such as the instruction that caused the exception and any data addresses associated with the exception. There are two pairs of these registers, one each for level 0 and level 1. They are the IXI0, IXA0 and IXI1, IXA1. The IXI0/1 save the instruction for levels 0 and 1, the IXA0/1 save the data addresses for level 0/1. The instruction is saved in the 32 bits of the register IXI0/1.

In addition, the processor may save additional information related to the exception in the 32 bits of the register IXIX0/1. This syndrome information is specific to the exception.

For DATA exceptions (DATA_ALIGN, DATA_MATCH) the following bits will be set:

- ld (bit 31): the exception was caused by a load instruction
- st (bit 30): the exception was caused by a store instruction
- ctl (bit 29): the exception was caused by a dctl instruction.

8.2.9 Summary of behavior

The following code summarizes the behavior of the interrupt subsystem

```
if INTERRUPT0 or ( XEN[0] = 1 and XPEN[0] = 1 and PSC[interrupt_x] = 1 ) then
  en_mc ← 1

for i in 1..31:
  if INTERRUPTi or ( XEN[i] = 1 and XPEN[i] = 1 and PSC[interrupt_x] = 1  ) then
    if xlv[i] = 0 then
      if PSC[interrupt_0] then
        en_0[i] ← 1
      else if xms[i] = 1 then
        en_mc[i] ← 1
      endif
    else
      if PSC[interrupt_1] then
        en_1[i] ← 1
      else if xms[i] = 1 then
        en_mc[i] ← 1
      endif
   endif

if PSC[interrupt_mc] = 1 then
  for i in 0..31:
    if en_mc[i] then
      XRM  ← PC
      XSM  ← PSC
      ilevel = mc
      inum = i
      goto handle_interrupt

if PSC[interrupt_0] = 1 then
  for i in 1..31:
    if en_0[i] then
```

```
    XR0H  ← PC
    XS0H  ← PSC

    inum   = i
    ilevel = 0
    goto handle_interrupt
   endif

 if PSC[interrupt_1] = 1 then
  for i in 1..31:
    if en_0[i] then
    XR1H  ← PC
    XS1H  ← PSC

    inum   = i
    ilevel = 1
    goto handle_interrupt
   endif

if instruction is trap N then
  XRT  ← PC
  XST  ← PSC
  PC   ← TXB + 256*N
  PSC[user] ← 0
  return


handle_interrupt:
if ilevel = mc then
  psc = PSCM
else
  psc = PSCH
endif

if ilevel = mc then
  psc[interrupt_mc] = 0
endif

if ilevel = mc or ilevel = 0 then
  psc[interrupt_0] = 0
endif

psc[interrupt_1] = 0

PSC ← psc

if ilevel = mc then
  PC   ← MXB + 256*inum
else
  PC   ← HXB + 256*inum
return
```

9 Appendix: Vector Exception Checks

The following table describes the conditions under which either a vector type exception or a vector length exception will be thrown.

In the table, two columns specify when a type exception is raised based on the values of the `it`/ `ft`/ `xit`/ `xft` fields. `h,w` are half word/word integer types. `h,s` are half precision/single precision floating-point types.

- If a vector instruction has an `it` field and the value does not match the allowable types, a vector type exception will be raised
- If a vector instruction has an `ft` field and the value does not match the allowable types, a vector type exception will be raised.
- If a vector instruction has an `xit` field and the integer type does not match the allowed value, then a vector type exception will be raised.
- If a vector instruction has an `xft` field and the floating-point type does not match the allowed value, then a vector type exception will be raised

The last column specifies the expression that must be satisfied by the values in the `$nsa` and `$nsb` register.

- `a` is the value of the `$nsa` register
- `b` is the value of the `$nsb` register
- `N` is the number of elements in the vector. It is based on the value in the `it`/ `ft`/ `xit`/ `xft` fields; for `h`/`h` it is 512, for `w`/`s` it is 256. If both `it`/`ft` and `xit`/ `xft` fields are present, then it is the minimum of the two numbers.

instruction	it/ft	xit/xft	Length
c2msum	hb		
vcsum	hb		
V2csum	hb		
v2mul	h		a<=N/2
v2muls	h		a<=N/2
vmul	h		a<=N
vmaxa	hw		a*b<=N && b > 1 && b <= N
vmina	hw		a*b<=N && b > 1 && b <= N
vadd	hw		a<=N
vand	hw		a<=N
vcand	hw		a<=N
vcmps	hw		a<=N
vcmpu	hw		a<=N
vdiff	hw		a<=N
vmax	hw		a<=N
vmin	hw		a<=N
vor	hw		a<=N
vshl	hw		a<=N
vshr	hw		a<=N
vshra	hw		a<=N
vsub	hw		a<=N
vxor	hw		a<=N
vbrd	hw		a<=N
vcopy	bhw		a<=N
vsel	hw		a<=N
vsetr	hw		a<=N,b<N
ldv	bhw		a<=N
stv	bhw		a<=N
vconcat	hw		a<=N
vdilv	hw		a*b<=N/2 && b > 0 && b <= N
vdupl	hw		a*b<=N && b > 1 && b <= N
vext	hw		a*b<=N && b > 1 && b <= N
vextb	hw		a*b<=N && b > 1 && b <= N
vilv	hw		a*b<=N/2 && b > 0 && b <= N
vins	hw		a*(b+1)<=N && b > 0 && b < N
vspread	hw		a*b<=N && b > 1 && b <= N
vtail	hw		a+b<=N && b <= N
rsetv	hw		b<N
vpack	hw		a<=N
vunpack	bh		a<=N/2
vmop			
msetr			
vcfm			
vctm			