MRISC32 Instruction Set Manual

Version 2021-*draft*

Marcus Geelnard, m@bitsnbites.eu

Preface

This document describes the MRISC32 instruction set architecture.

Contents

Chapter 1

Introduction

1.1 Overview

MRISC32 is an open and free instruction set architecture (ISA).

It is a RISC style load-store vector architecture that is designed to be suitable for a wide range of performance levels.

> **TODO**
>
> *Add wording about goals.*

1.2 Data types

1.2.1 Size types

The following types define a size without mandating any particular interpretation of the data:

Name	Size
word	32 bits
half-word	16 bits
byte	8 bits

1.2.2 Integer types

Signed integer types are represented in two's complement form.

Name	Size	Meaning
int32	32	Signed 32-bit integer
uint32	32	Unsigned 32-bit integer
int16	16	Signed 16-bit integer
uint16	16	Unsigned 16-bit integer
int8	8	Signed 8-bit integer
uint8	8	Unsigned 8-bit integer

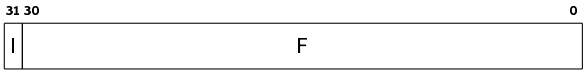
1.2.3 Fixed point types

For some operations the fixed point Q number format is used, in which the most significant bit is the integer/sign bit, and the rest of the bits are the fractional bits.

Q31

Q31 is a 32-bit signed fixed point number with 31 fractional bits.

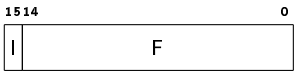
The value of a Q31 number is obtained by interpreting the bit vector as a two's complement signed integer multiplied by 2^{-31}.



Q15

Q15 is a 16-bit signed fixed point number with 15 fractional bits.

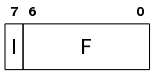
The value of a Q15 number is obtained by interpreting the bit vector as a two's complement signed integer multiplied by 2^{-15}.



Q7

Q7 is an 8-bit signed fixed point number with 7 fractional bits.

The value of a Q7 number is obtained by interpreting the bit vector as a two's complement signed integer multiplied by 2^{-7}.



1.2.4 Floating-point types

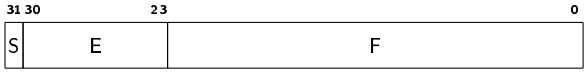
Name	Size	Meaning
float32	32	Single precision binary floating-point number
float16	16	Half precision binary floating-point number
float8	8	Quarter precision binary floating-point number

float32

The float32 type uses one sign bit (S), eight exponent bits (E) and 23 fractional bits (F)[1].

The significand has an implicit leading bit (to the left of the binary point) with value 1, giving 24 effective significand bits.

The exponent bias is 127.

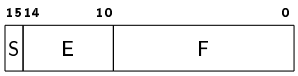


float16

The float16 type uses one sign bit (S), five exponent bits (E) and ten fractional bits (F)[2].

The significand has an implicit leading bit (to the left of the binary point) with value 1, giving eleven effective significand bits.

The exponent bias is 15.



[1] The float32 type uses the same format and interpretation as IEEE 754-2008 binary32

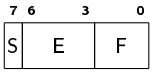
[2] The float16 type uses the same format and interpretation as IEEE 754-2008 binary16

float8

The float8 type uses one sign bit (S), four exponent bits (E) and three fractional bits (F).

The significand has an implicit leading bit (to the left of the binary point) with value 1, giving four effective significand bits.

The exponent bias is 7.



1.3 Pseudocode syntax

The operation that an instruction performs is described using pseudocode.

1.3.1 Pseudocode scope

The pseudocode for each instruction shall be regarded as a function that is executed for each chunk of each element of the operation.

For a scalar operation, there is only a single element.

For a vector operation, the number of elements is dictated by the vector operation.

The number of chunks and the size of each chunk is dictated by the packed operation mode.

> **TODO**
>
> *Describe different types that are used in the pseudocode, such as "bit vector", "integer", "unsigned integer" and "float"?*
>
> *Come up with a better name than "chunk"?*

1.3.2 Notation

The following notation is used in the pseudocode that describes the operation of an instruction:

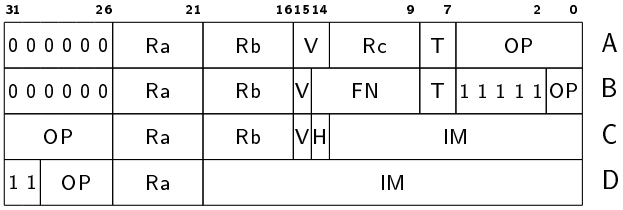
Notation	Meaning
x<k>	Bit k of bit vector x
MEM[x,N]	N consecutive bytes in memory starting at address x, interpreted as an $8 \times N$-bit vector with little endian storage
a	1st instruction operand
b	2nd instruction operand
c	3rd instruction operand
IM	IM field of the instruction word
T	T field of the instruction word
V	Vector mode (two bits)
bits	Chunk size, in bits
scale	Scale factor according to the T field (1 for format C instructions)
i	Vector element number
$\leftarrow$	Assignment
+	Addition
-	Subtraction
$\times$	Multiplication
/	Division
%	Remainder of integer division
$=$	Equal
$\neq$	Not equal
$\leq$	Less than or equal
$\geq$	Greater than or equal
$\neg$	Logical not
$\wedge$	Logical or
$\vee$	Logical and
~	Bitwise not
|	Bitwise or
&	Bitwise and
^	Bitwise xor
<<	Bit shift left
>>	Bit shift right (preserve sign for signed integer types)
ones(N)	Bit vector of N 1-bits
zeros(N)	Bit vector of N 0-bits
int(x)	Interpret bit vector x as a two's complement signed integer
uint(x)	Interpret bit vector x as an unsigned integer
float(x)	Interpret bit vector x as a floating-point number
$\log_2$(x)	Binary logarithm of x
max(x,y)	Maximum value of x and y
min(x,y)	Minimum value of x and y
sat(x,N)	Saturate integer x to the range $[-2^{N-1}, 2^{N-1})$
satu(x,N)	Saturate integer x to the range $[0, 2^N)$
isnan(x)	True if x is not a number (NaN)

1.4 Assembler syntax

TBD

1.5 Instruction formats

All instructions are encoded in 32 bits. There are four different encoding formats, A, B, C and D, that mainly differ in the number and kinds of instruction operands.



1.5.1 Instruction word fields

The field names that are used in the instruction format descriptions are listed in the table below:

Name	Meaning
OP	Operation
FN	Function (extended operation)
V	Vector Mode
T	Type
Ra	Destination/source register number (0-31)
Rb	Source register number (0-31)
Rc	Source register number (0-31)
H	Immediate Hi/Lo flag
IM	Immediate value

Not all field types appear in all instruction formats.

The OP field in combination with the FN field (where applicable) is the main identification of the instruction, and dictates what operation the instruction shall perform.

The V field defines the scalar/vector configuration of the operands. The scalar/vector operand configuration is a two-bit identifier. When only one bit is provided by the V field, that bit is used as the most significant bit of the identifier, and the least significant bit is implicitly zero.

Operand types (S for scalar, V for vector) for each operand positions relates to the V identifier as follows

(note that load/store instructions always interpret the second operand - i.e. the base address - as a scalar):

V	Default	Load/store
00	S,S[,S]	S,S,S
10	V,V[,S]	V,S,S
11	V,V,V	V,S,V
01	V,V,fold(V)	V,S,fold(V)

The register fields Ra, Rb and Rc refer to one scalar or vector register each, according to the OP and V fields. For instance if a register operand refers to a vector register, and the corresponding R-field has the value 21, then the register operand is V21.

The first register operand, Ra, can be a source or a destination register depending on the instruction, while Rb and Rc are always source registers.

The T field further defines the instruction. For most instructions it defines the packed data type that is to be used (for packed operations). For load/store instructions it defines a scaling factor for the register offset operand (i.e. the third operand):

T	Default	Load/store
00	One 32-bit word	*1
01	Four 8-bit bytes	*2
10	Two 16-bit half-words	*4
11	(reserved)	*8

The IM field provides an immediate value. The size of the IM field depends on the instruction format, and the interpretation of the field further depends on the OP and the H fields.

The H field describes how to interpret a 14-bit IM field:

- When H=0, the IM value is sign-extended to 32 bits.
- When H=1, the IM value is shifted to the left 18 position, and the least significant bit of the IM field is duplicated to the 18 least significant bits of the final immediate value.

1.5.2 Format A

Format A instructions are used for instructions that require three register operands, and support both vector and packed operations.

1.5.3 Format B

Format B instructions are used for instructions that only require two register operands (for instance unary operations). Both vector and packed operations are supported.

1.5.4 Format C

Format C instructions are used for instructions that require two register operands and one immediate operand. Vector operations are supported (but not packed operations).

In general each format C instruction has a corresponding format A encoding with the same value of the OP field.

1.5.5 Format D

Format D is used for instructions that need to be able to express large immediate values.

Chapter 2

Programming model

2.1 Architecture profiles

TODO

Describe the base ISA and extensions, i.e. roughly:

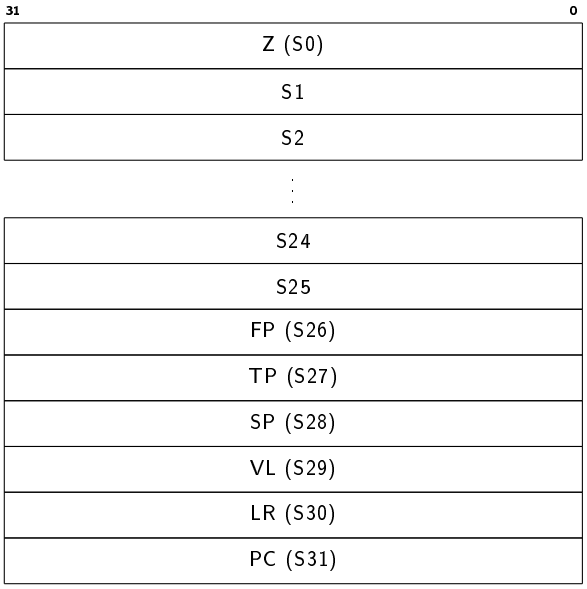
- *Support for vector operations.*
- *Support for packed operations.*
- *Support for floating-point operations.*
- *Support for saturating and halving arithmetic.*

We should also define 2-4 profiles that an implementation must comply to (to minimize fragmentation). This is more to convey the intent of the ISA than anything else.

We also need to document which instructions are not part of the base ISA, e.g. by tagging them in the instruction database.

2.2 Scalar registers

There are 32 scalar registers, each 32 bits wide.



2.2.1 The Z register

Z is a read-only register that is always zero. Writing to the Z register has no effect.

2.2.2 The VL register

VL is the vector length register, which defines the length of vector operations.

TODO

Describe how the VL register affects vector operations.

2.2.3 The LR register

LR is the link register, which contains the return address for subroutines.

2.2.4 The PC register

The PC is a read-only register that contains the memory address of the current instruction. Writing to the PC register has no effect.

2.2.5 FP, TP and SP

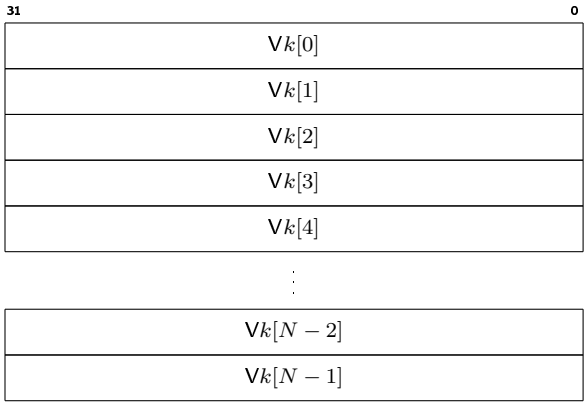
The scalar registers FP, TP and SP are aliases for S26, S27 and S28, respectively. They have no special meaning in hardware, but it is recommended that they are used as follows:

Name	Description
FP	Frame pointer
TP	Thread pointer (for thread local storage)
SP	Stack pointer

2.3 Vector registers

There are 32 vector registers: VZ, V1, V2, ..., V30, V31.

Each register, Vk, consists of N 32-bit elements, where N is implementation defined (N must be a power of two, and at least 16):



2.3.1 The VZ register

VZ is a read-only register with all vector elements set to zero. Writing to the VZ register has no effect.

2.4 Vector operation

TBD

2.5 Packed data operation

TBD

2.6 Floating-point operation

The MRISC32 ISA supports a subset of the 2008 IEEE-754 floating-point standard [1].

TBD

2.7 Memory addressing

TBD

2.8 Exceptions

TBD

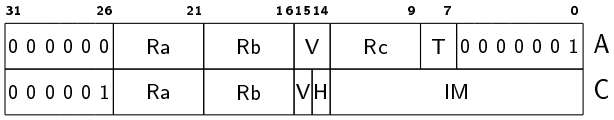
Chapter 3

Instructions

3.1 Load and store

3.1.1 LDB

Load and sign extend a byte (8 bits).

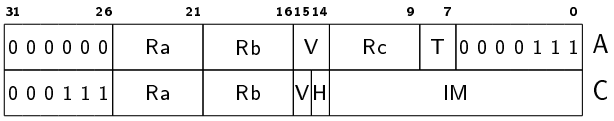



```
if V = 10 then
  adr ← int(b) + int(c) × i × scale
else
  adr ← int(b) + int(c) × scale
a ← int(MEM[adr,1])
```

Fmt	V	T	Assembler	
A	00	00	LDB	Sa, Sb, Sc
A	00	01	LDB	Sa, Sb, Sc*2
A	00	10	LDB	Sa, Sb, Sc*4
A	00	11	LDB	Sa, Sb, Sc*8
A	10	00	LDB	Va, Sb, Sc
A	10	01	LDB	Va, Sb, Sc*2
A	10	10	LDB	Va, Sb, Sc*4
A	10	11	LDB	Va, Sb, Sc*8
A	11	00	LDB	Va, Sb, Vc
A	11	01	LDB	Va, Sb, Vc*2
A	11	10	LDB	Va, Sb, Vc*4
A	11	11	LDB	Va, Sb, Vc*8
A	01	00	LDB/F	Va, Sb, Vc
A	01	01	LDB/F	Va, Sb, Vc*2
A	01	10	LDB/F	Va, Sb, Vc*4
A	01	11	LDB/F	Va, Sb, Vc*8
C	00	00	LDB	Sa, Sb, #ext14(H,IM)
C	10	00	LDB	Va, Sb, #ext14(H,IM)

3.1.2 LDEA

Load effective address.



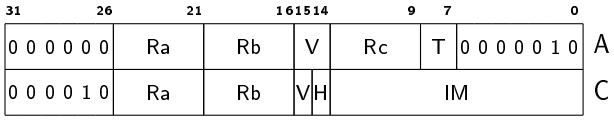

```
if V = 10 then
  a ← int(b) + int(c) × i × scale
else
  a ← int(b) + int(c) × scale
```

Fmt	V	T	Assembler	
A	00	00	LDEA	Sa, Sb, Sc
A	00	01	LDEA	Sa, Sb, Sc*2
A	00	10	LDEA	Sa, Sb, Sc*4
A	00	11	LDEA	Sa, Sb, Sc*8
A	10	00	LDEA	Va, Sb, Sc
A	10	01	LDEA	Va, Sb, Sc*2
A	10	10	LDEA	Va, Sb, Sc*4
A	10	11	LDEA	Va, Sb, Sc*8
A	11	00	LDEA	Va, Sb, Vc
A	11	01	LDEA	Va, Sb, Vc*2
A	11	10	LDEA	Va, Sb, Vc*4
A	11	11	LDEA	Va, Sb, Vc*8
A	01	00	LDEA/F	Va, Sb, Vc
A	01	01	LDEA/F	Va, Sb, Vc*2
A	01	10	LDEA/F	Va, Sb, Vc*4
A	01	11	LDEA/F	Va, Sb, Vc*8
C	00	00	LDEA	Sa, Sb, #ext14(H,IM)
C	10	00	LDEA	Va, Sb, #ext14(H,IM)

> **Note**
>
> When the target operand is a vector register, LDEA can be used for constructing strides. For instance LDEA V1,Z,#3 will assign the vector [0,3,6,9,...] to register V1.

3.1.3 LDH

Load and sign extend a half-word (16 bits).

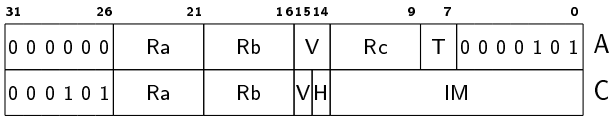



```
if V = 10 then
  adr ← int(b) + int(c) × i × scale
else
  adr ← int(b) + int(c) × scale
a ← int(MEM[adr,2])
```

Fmt	V	T	Assembler
A	00	00	LDH Sa, Sb, Sc
A	00	01	LDH Sa, Sb, Sc*2
A	00	10	LDH Sa, Sb, Sc*4
A	00	11	LDH Sa, Sb, Sc*8
A	10	00	LDH Va, Sb, Sc
A	10	01	LDH Va, Sb, Sc*2
A	10	10	LDH Va, Sb, Sc*4
A	10	11	LDH Va, Sb, Sc*8
A	11	00	LDH Va, Sb, Vc
A	11	01	LDH Va, Sb, Vc*2
A	11	10	LDH Va, Sb, Vc*4
A	11	11	LDH Va, Sb, Vc*8
A	01	00	LDH/F Va, Sb, Vc
A	01	01	LDH/F Va, Sb, Vc*2
A	01	10	LDH/F Va, Sb, Vc*4
A	01	11	LDH/F Va, Sb, Vc*8
C	00	00	LDH Sa, Sb, #ext14(H,IM)
C	10	00	LDH Va, Sb, #ext14(H,IM)

3.1.4 LDUB

Load and zero extend a byte (8 bits).

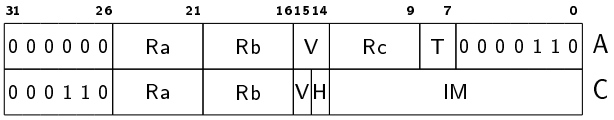



```
if V = 10 then
  adr ← int(b) + int(c) × i × scale
else
  adr ← int(b) + int(c) × scale
a ← uint(MEM[adr,1])
```

Fmt	V	T	Assembler
A	00	00	LDUB Sa, Sb, Sc
A	00	01	LDUB Sa, Sb, Sc*2
A	00	10	LDUB Sa, Sb, Sc*4
A	00	11	LDUB Sa, Sb, Sc*8
A	10	00	LDUB Va, Sb, Sc
A	10	01	LDUB Va, Sb, Sc*2
A	10	10	LDUB Va, Sb, Sc*4
A	10	11	LDUB Va, Sb, Sc*8
A	11	00	LDUB Va, Sb, Vc
A	11	01	LDUB Va, Sb, Vc*2
A	11	10	LDUB Va, Sb, Vc*4
A	11	11	LDUB Va, Sb, Vc*8
A	01	00	LDUB/F Va, Sb, Vc
A	01	01	LDUB/F Va, Sb, Vc*2
A	01	10	LDUB/F Va, Sb, Vc*4
A	01	11	LDUB/F Va, Sb, Vc*8
C	00	00	LDUB Sa, Sb, #ext14(H,IM)
C	10	00	LDUB Va, Sb, #ext14(H,IM)

3.1.5 LDUH

Load and zero extend a half-word (16 bits).



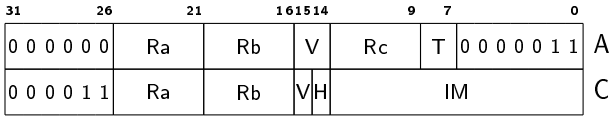

```
if V = 10 then
  adr ← int(b) + int(c) × i × scale
else
  adr ← int(b) + int(c) × scale
a ← uint(MEM[adr,2])
```

Fmt	V	T	Assembler
A	00	00	LDUH Sa, Sb, Sc
A	00	01	LDUH Sa, Sb, Sc*2
A	00	10	LDUH Sa, Sb, Sc*4
A	00	11	LDUH Sa, Sb, Sc*8
A	10	00	LDUH Va, Sb, Sc
A	10	01	LDUH Va, Sb, Sc*2
A	10	10	LDUH Va, Sb, Sc*4
A	10	11	LDUH Va, Sb, Sc*8
A	11	00	LDUH Va, Sb, Vc
A	11	01	LDUH Va, Sb, Vc*2
A	11	10	LDUH Va, Sb, Vc*4
A	11	11	LDUH Va, Sb, Vc*8
A	01	00	LDUH/F Va, Sb, Vc
A	01	01	LDUH/F Va, Sb, Vc*2
A	01	10	LDUH/F Va, Sb, Vc*4
A	01	11	LDUH/F Va, Sb, Vc*8
C	00	00	LDUH Sa, Sb, #ext14(H,IM)
C	10	00	LDUH Va, Sb, #ext14(H,IM)

Fmt	V	T	Assembler
A	00	00	LDW Sa, Sb, Sc
A	00	01	LDW Sa, Sb, Sc*2
A	00	10	LDW Sa, Sb, Sc*4
A	00	11	LDW Sa, Sb, Sc*8
A	10	00	LDW Va, Sb, Sc
A	10	01	LDW Va, Sb, Sc*2
A	10	10	LDW Va, Sb, Sc*4
A	10	11	LDW Va, Sb, Sc*8
A	11	00	LDW Va, Sb, Vc
A	11	01	LDW Va, Sb, Vc*2
A	11	10	LDW Va, Sb, Vc*4
A	11	11	LDW Va, Sb, Vc*8
A	01	00	LDW/F Va, Sb, Vc
A	01	01	LDW/F Va, Sb, Vc*2
A	01	10	LDW/F Va, Sb, Vc*4
A	01	11	LDW/F Va, Sb, Vc*8
C	00	00	LDW Sa, Sb, #ext14(H,IM)
C	10	00	LDW Va, Sb, #ext14(H,IM)

3.1.6 LDW

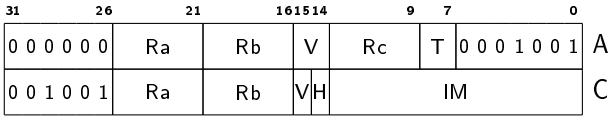
Load a word (32 bits).




```
if V = 10 then
  adr ← int(b) + int(c) × i × scale
else
  adr ← int(b) + int(c) × scale
a ← int(MEM[adr,4])
```

3.1.7 STB

Store a byte (8 bits).



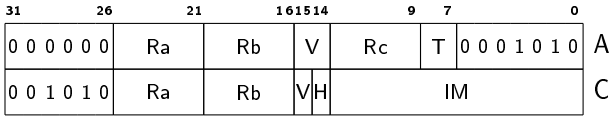

```
if V = 10 then
  adr ← int(b) + int(c) × i × scale
else
  adr ← int(b) + int(c) × scale
MEM[adr,1] ← a
```

Fmt	V	T	Assembler	
A	00	00	STB	Sa, Sb, Sc
A	00	01	STB	Sa, Sb, Sc*2
A	00	10	STB	Sa, Sb, Sc*4
A	00	11	STB	Sa, Sb, Sc*8
A	10	00	STB	Va, Sb, Sc
A	10	01	STB	Va, Sb, Sc*2
A	10	10	STB	Va, Sb, Sc*4
A	10	11	STB	Va, Sb, Sc*8
A	11	00	STB	Va, Sb, Vc
A	11	01	STB	Va, Sb, Vc*2
A	11	10	STB	Va, Sb, Vc*4
A	11	11	STB	Va, Sb, Vc*8
A	01	00	STB/F	Va, Sb, Vc
A	01	01	STB/F	Va, Sb, Vc*2
A	01	10	STB/F	Va, Sb, Vc*4
A	01	11	STB/F	Va, Sb, Vc*8
C	00	00	STB	Sa, Sb, #ext14(H,IM)
C	10	00	STB	Va, Sb, #ext14(H,IM)

Fmt	V	T	Assembler	
A	00	00	STH	Sa, Sb, Sc
A	00	01	STH	Sa, Sb, Sc*2
A	00	10	STH	Sa, Sb, Sc*4
A	00	11	STH	Sa, Sb, Sc*8
A	10	00	STH	Va, Sb, Sc
A	10	01	STH	Va, Sb, Sc*2
A	10	10	STH	Va, Sb, Sc*4
A	10	11	STH	Va, Sb, Sc*8
A	11	00	STH	Va, Sb, Vc
A	11	01	STH	Va, Sb, Vc*2
A	11	10	STH	Va, Sb, Vc*4
A	11	11	STH	Va, Sb, Vc*8
A	01	00	STH/F	Va, Sb, Vc
A	01	01	STH/F	Va, Sb, Vc*2
A	01	10	STH/F	Va, Sb, Vc*4
A	01	11	STH/F	Va, Sb, Vc*8
C	00	00	STH	Sa, Sb, #ext14(H,IM)
C	10	00	STH	Va, Sb, #ext14(H,IM)

3.1.8 STH

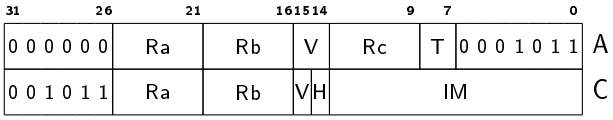
Store a half-word (16 bits).




```
if V = 10 then
  adr ← int(b) + int(c) × i × scale
else
  adr ← int(b) + int(c) × scale
MEM[adr,2] ← a
```

3.1.9 STW

Store a word (32 bits).



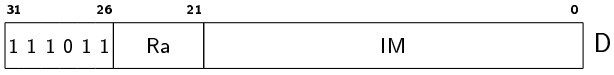

```
if V = 10 then
  adr ← int(b) + int(c) × i × scale
else
  adr ← int(b) + int(c) × scale
MEM[adr,4] ← a
```

Fmt	V	T	Assembler	
A	00	00	STW	Sa, Sb, Sc
A	00	01	STW	Sa, Sb, Sc*2
A	00	10	STW	Sa, Sb, Sc*4
A	00	11	STW	Sa, Sb, Sc*8
A	10	00	STW	Va, Sb, Sc
A	10	01	STW	Va, Sb, Sc*2
A	10	10	STW	Va, Sb, Sc*4
A	10	11	STW	Va, Sb, Sc*8
A	11	00	STW	Va, Sb, Vc
A	11	01	STW	Va, Sb, Vc*2
A	11	10	STW	Va, Sb, Vc*4
A	11	11	STW	Va, Sb, Vc*8
A	01	00	STW/F	Va, Sb, Vc
A	01	01	STW/F	Va, Sb, Vc*2
A	01	10	STW/F	Va, Sb, Vc*4
A	01	11	STW/F	Va, Sb, Vc*8
C	00	00	STW	Sa, Sb, #ext14(H,IM)
C	10	00	STW	Va, Sb, #ext14(H,IM)

3.2 Load immediate

3.2.1 LDHI

Load high immediate value.




```
a ← int(IM) << 11
```

Assembler

```
LDHI    Sa, #ext21hi(IM)
```

3.2.2 LDHIO

Load high immediate value with low ones.

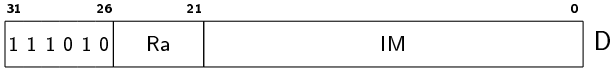

```
a ← (int(IM) << 11) | ones(11)
```

Assembler

```
LDHIO   Sa, #ext21hio(IM)
```

3.2.3 LDLI

Load low immediate value.




```
a ← int(IM)
```

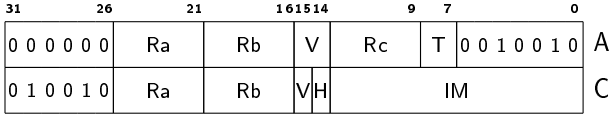
Assembler

```
LDLI    Sa, #ext21(IM)
```

3.3 Bitwise logic

3.3.1 AND

Compute the bitwise and of two integer operands.

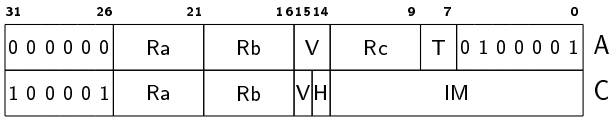



```
a ← b & c
```

Fmt	V	Assembler	
A	00	AND	Sa, Sb, Sc
A	10	AND	Va, Vb, Sc
A	11	AND	Va, Vb, Vc
A	01	AND/F	Va, Vb, Vc
C	00	AND	Sa, Sb, #ext14(H,IM)
C	10	AND	Va, Vb, #ext14(H,IM)

3.3.2 ASR

Arithmetically shift an integer operand to the right the number of bit positions given by another integer operand. Only the lowest log2(bits) bits of the shift operand are used.

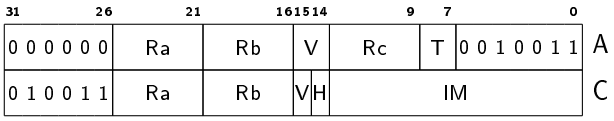



```
a ← int(b) >> int(c & ones(log2(bits)))
```

Fmt	V	T	Assembler	
A	00	00	ASR	Sa, Sb, Sc
A	00	01	ASR.B	Sa, Sb, Sc
A	00	10	ASR.H	Sa, Sb, Sc
A	10	00	ASR	Va, Vb, Sc
A	10	01	ASR.B	Va, Vb, Sc
A	10	10	ASR.H	Va, Vb, Sc
A	11	00	ASR	Va, Vb, Vc
A	11	01	ASR.B	Va, Vb, Vc
A	11	10	ASR.H	Va, Vb, Vc
A	01	00	ASR/F	Va, Vb, Vc
A	01	01	ASR.B/F	Va, Vb, Vc
A	01	10	ASR.H/F	Va, Vb, Vc
C	00	00	ASR	Sa, Sb, #ext14(H,IM)
C	10	00	ASR	Va, Vb, #ext14(H,IM)

3.3.3 BIC

Compute the bitwise clear of two integer operands.

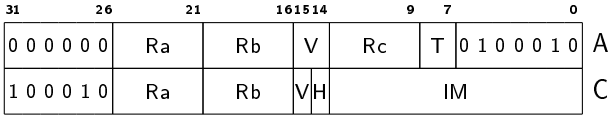



```
a ← b & ~c
```

Fmt	V	Assembler	
A	00	BIC	Sa, Sb, Sc
A	10	BIC	Va, Vb, Sc
A	11	BIC	Va, Vb, Vc
A	01	BIC/F	Va, Vb, Vc
C	00	BIC	Sa, Sb, #ext14(H,IM)
C	10	BIC	Va, Vb, #ext14(H,IM)

3.3.4 LSL

Logically shift an integer operand to the left the number of bit positions given by another integer operand. Only the lowest log2(bits) bits of the shift operand are used.

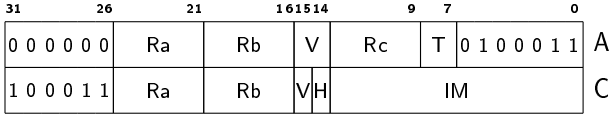



```
a ← uint(b) << int(c & ones(log2(bits)))
```

Fmt	V	T	Assembler	
A	00	00	LSL	Sa, Sb, Sc
A	00	01	LSL.B	Sa, Sb, Sc
A	00	10	LSL.H	Sa, Sb, Sc
A	10	00	LSL	Va, Vb, Sc
A	10	01	LSL.B	Va, Vb, Sc
A	10	10	LSL.H	Va, Vb, Sc
A	11	00	LSL	Va, Vb, Vc
A	11	01	LSL.B	Va, Vb, Vc
A	11	10	LSL.H	Va, Vb, Vc
A	01	00	LSL/F	Va, Vb, Vc
A	01	01	LSL.B/F	Va, Vb, Vc
A	01	10	LSL.H/F	Va, Vb, Vc
C	00	00	LSL	Sa, Sb, #ext14(H,IM)
C	10	00	LSL	Va, Vb, #ext14(H,IM)

3.3.5 LSR

Logically shift an integer operand to the right the number of bit positions given by another integer operand. Only the lowest log2(bits) bits of the shift operand are used.

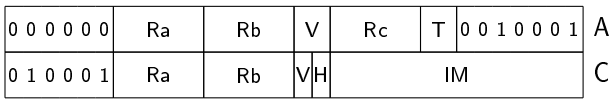



```
a ← uint(b) >> int(c & ones(log2(bits)))
```

Fmt	V	T	Assembler
A	00	00	LSR Sa, Sb, Sc
A	00	01	LSR.B Sa, Sb, Sc
A	00	10	LSR.H Sa, Sb, Sc
A	10	00	LSR Va, Vb, Sc
A	10	01	LSR.B Va, Vb, Sc
A	10	10	LSR.H Va, Vb, Sc
A	11	00	LSR Va, Vb, Vc
A	11	01	LSR.B Va, Vb, Vc
A	11	10	LSR.H Va, Vb, Vc
A	01	00	LSR/F Va, Vb, Vc
A	01	01	LSR.B/F Va, Vb, Vc
A	01	10	LSR.H/F Va, Vb, Vc
C	00	00	LSR Sa, Sb, #ext14(H,IM)
C	10	00	LSR Va, Vb, #ext14(H,IM)

3.3.6 NOR

Compute the bitwise inverse or of two integer operands.

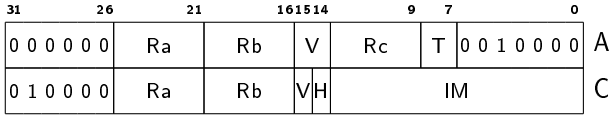



```
a ← ~(b | c)
```

Fmt	V	Assembler
A	00	NOR Sa, Sb, Sc
A	10	NOR Va, Vb, Sc
A	11	NOR Va, Vb, Vc
A	01	NOR/F Va, Vb, Vc
C	00	NOR Sa, Sb, #ext14(H,IM)
C	10	NOR Va, Vb, #ext14(H,IM)

3.3.7 OR

Compute the bitwise or of two integer operands.

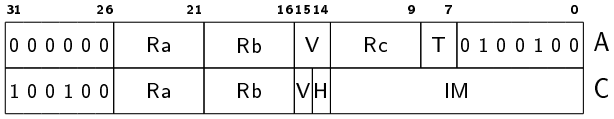



```
a ← b | c
```

Fmt	V	Assembler
A	00	OR Sa, Sb, Sc
A	10	OR Va, Vb, Sc
A	11	OR Va, Vb, Vc
A	01	OR/F Va, Vb, Vc
C	00	OR Sa, Sb, #ext14(H,IM)
C	10	OR Va, Vb, #ext14(H,IM)

3.3.8 SEL

Bitwise select.

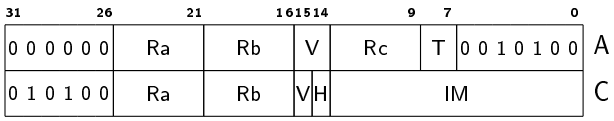



```
if T = 00 then
  a ← (b & a) | (c & ~a)
else if T = 01 then
  a ← (c & a) | (b & ~a)
else if T = 10 then
  a ← (a & b) | (c & ~b)
else if T = 11 then
  a ← (c & b) | (a & ~b)
```

Fmt	V	T	Assembler
A	00	00	SEL Sa, Sb, Sc
A	00	01	SEL_1 Sa, Sb, Sc
A	00	10	SEL_2 Sa, Sb, Sc
A	00	11	SEL_3 Sa, Sb, Sc
A	10	00	SEL Va, Vb, Sc
A	10	01	SEL_1 Va, Vb, Sc
A	10	10	SEL_2 Va, Vb, Sc
A	10	11	SEL_3 Va, Vb, Sc
A	11	00	SEL Va, Vb, Vc
A	11	01	SEL_1 Va, Vb, Vc
A	11	10	SEL_2 Va, Vb, Vc
A	11	11	SEL_3 Va, Vb, Vc
A	01	00	SEL/F Va, Vb, Vc
A	01	01	SEL_1/F Va, Vb, Vc
A	01	10	SEL_2/F Va, Vb, Vc
A	01	11	SEL_3/F Va, Vb, Vc
C	00	00	SEL Sa, Sb, #ext14(H,IM)
C	10	00	SEL Va, Vb, #ext14(H,IM)

3.3.9 XOR

Compute the bitwise exclusive or of two integer operands.



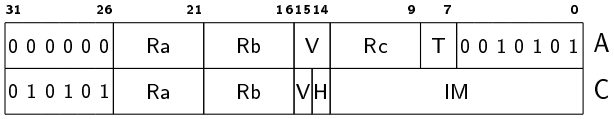
a ← b ^ c

Fmt	V	Assembler
A	00	XOR Sa, Sb, Sc
A	10	XOR Va, Vb, Sc
A	11	XOR Va, Vb, Vc
A	01	XOR/F Va, Vb, Vc
C	00	XOR Sa, Sb, #ext14(H,IM)
C	10	XOR Va, Vb, #ext14(H,IM)

3.4 Integer arithmetic

3.4.1 ADD

Compute the sum of two integer operands.

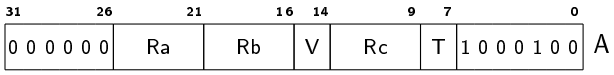



```
a ← int(b) + int(c)
```

Fmt	V	T	Assembler
A	00	00	ADD Sa, Sb, Sc
A	00	01	ADD.B Sa, Sb, Sc
A	00	10	ADD.H Sa, Sb, Sc
A	10	00	ADD Va, Vb, Sc
A	10	01	ADD.B Va, Vb, Sc
A	10	10	ADD.H Va, Vb, Sc
A	11	00	ADD Va, Vb, Vc
A	11	01	ADD.B Va, Vb, Vc
A	11	10	ADD.H Va, Vb, Vc
A	01	00	ADD/F Va, Vb, Vc
A	01	01	ADD.B/F Va, Vb, Vc
A	01	10	ADD.H/F Va, Vb, Vc
C	00	00	ADD Sa, Sb, #ext14(H,IM)
C	10	00	ADD Va, Vb, #ext14(H,IM)

3.4.2 DIV

Compute the quotient of two signed integer operands.

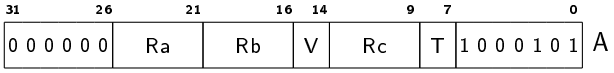



```
a ← int(b) / int(c)
```

V	T	Assembler
00	00	DIV Sa, Sb, Sc
00	01	DIV.B Sa, Sb, Sc
00	10	DIV.H Sa, Sb, Sc
10	00	DIV Va, Vb, Sc
10	01	DIV.B Va, Vb, Sc
10	10	DIV.H Va, Vb, Sc
11	00	DIV Va, Vb, Vc
11	01	DIV.B Va, Vb, Vc
11	10	DIV.H Va, Vb, Vc
01	00	DIV/F Va, Vb, Vc
01	01	DIV.B/F Va, Vb, Vc
01	10	DIV.H/F Va, Vb, Vc

3.4.3 DIVU

Compute the quotient of two unsigned integer operands.

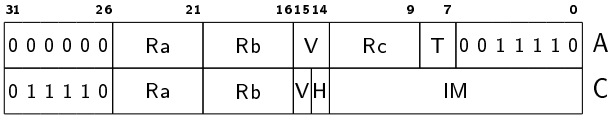



```
a ← uint(b) / uint(c)
```

V	T	Assembler
00	00	DIVU Sa, Sb, Sc
00	01	DIVU.B Sa, Sb, Sc
00	10	DIVU.H Sa, Sb, Sc
10	00	DIVU Va, Vb, Sc
10	01	DIVU.B Va, Vb, Sc
10	10	DIVU.H Va, Vb, Sc
11	00	DIVU Va, Vb, Vc
11	01	DIVU.B Va, Vb, Vc
11	10	DIVU.H Va, Vb, Vc
01	00	DIVU/F Va, Vb, Vc
01	01	DIVU.B/F Va, Vb, Vc
01	10	DIVU.H/F Va, Vb, Vc

3.4.4 MAX

Return the maximum value of two signed integer operands.

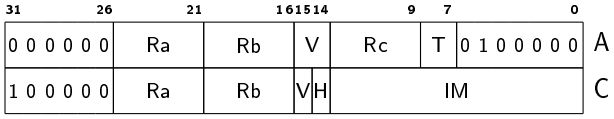



```
a ← max(int(b), int(c))
```

Fmt	V	T	Assembler
A	00	00	MAX Sa, Sb, Sc
A	00	01	MAX.B Sa, Sb, Sc
A	00	10	MAX.H Sa, Sb, Sc
A	10	00	MAX Va, Vb, Sc
A	10	01	MAX.B Va, Vb, Sc
A	10	10	MAX.H Va, Vb, Sc
A	11	00	MAX Va, Vb, Vc
A	11	01	MAX.B Va, Vb, Vc
A	11	10	MAX.H Va, Vb, Vc
A	01	00	MAX/F Va, Vb, Vc
A	01	01	MAX.B/F Va, Vb, Vc
A	01	10	MAX.H/F Va, Vb, Vc
C	00	00	MAX Sa, Sb, #ext14(H,IM)
C	10	00	MAX Va, Vb, #ext14(H,IM)

3.4.5 MAXU

Return the maximum value of two unsigned integer operands.

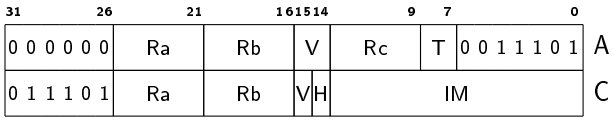



```
a ← max(uint(b), uint(c))
```

Fmt	V	T	Assembler
A	00	00	MAXU Sa, Sb, Sc
A	00	01	MAXU.B Sa, Sb, Sc
A	00	10	MAXU.H Sa, Sb, Sc
A	10	00	MAXU Va, Vb, Sc
A	10	01	MAXU.B Va, Vb, Sc
A	10	10	MAXU.H Va, Vb, Sc
A	11	00	MAXU Va, Vb, Vc
A	11	01	MAXU.B Va, Vb, Vc
A	11	10	MAXU.H Va, Vb, Vc
A	01	00	MAXU/F Va, Vb, Vc
A	01	01	MAXU.B/F Va, Vb, Vc
A	01	10	MAXU.H/F Va, Vb, Vc
C	00	00	MAXU Sa, Sb, #ext14(H,IM)
C	10	00	MAXU Va, Vb, #ext14(H,IM)

3.4.6 MIN

Return the minimum value of two signed integer operands.

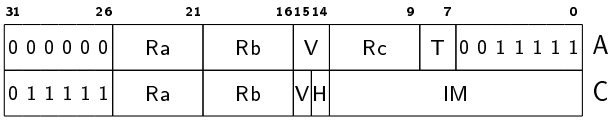



```
a ← min(int(b), int(c))
```

Fmt	V	T	Assembler
A	00	00	MIN Sa, Sb, Sc
A	00	01	MIN.B Sa, Sb, Sc
A	00	10	MIN.H Sa, Sb, Sc
A	10	00	MIN Va, Vb, Sc
A	10	01	MIN.B Va, Vb, Sc
A	10	10	MIN.H Va, Vb, Sc
A	11	00	MIN Va, Vb, Vc
A	11	01	MIN.B Va, Vb, Vc
A	11	10	MIN.H Va, Vb, Vc
A	01	00	MIN/F Va, Vb, Vc
A	01	01	MIN.B/F Va, Vb, Vc
A	01	10	MIN.H/F Va, Vb, Vc
C	00	00	MIN Sa, Sb, #ext14(H,IM)
C	10	00	MIN Va, Vb, #ext14(H,IM)

3.4.7 MINU

Return the minimum value of two unsigned integer operands.

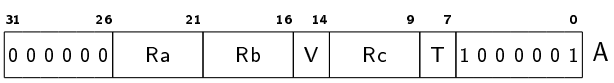



```
a ← min(uint(b), uint(c))
```

Fmt	V	T	Assembler
A	00	00	MINU Sa, Sb, Sc
A	00	01	MINU.B Sa, Sb, Sc
A	00	10	MINU.H Sa, Sb, Sc
A	10	00	MINU Va, Vb, Sc
A	10	01	MINU.B Va, Vb, Sc
A	10	10	MINU.H Va, Vb, Sc
A	11	00	MINU Va, Vb, Vc
A	11	01	MINU.B Va, Vb, Vc
A	11	10	MINU.H Va, Vb, Vc
A	01	00	MINU/F Va, Vb, Vc
A	01	01	MINU.B/F Va, Vb, Vc
A	01	10	MINU.H/F Va, Vb, Vc
C	00	00	MINU Sa, Sb, #ext14(H,IM)
C	10	00	MINU Va, Vb, #ext14(H,IM)

3.4.8 MUL

Compute the product of two integer operands.

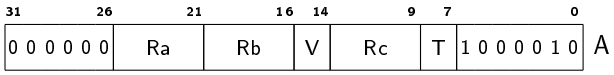


a ← int(b) × int(c)

V	T	Assembler
00	00	MUL Sa, Sb, Sc
00	01	MUL.B Sa, Sb, Sc
00	10	MUL.H Sa, Sb, Sc
10	00	MUL Va, Vb, Sc
10	01	MUL.B Va, Vb, Sc
10	10	MUL.H Va, Vb, Sc
11	00	MUL Va, Vb, Vc
11	01	MUL.B Va, Vb, Vc
11	10	MUL.H Va, Vb, Vc
01	00	MUL/F Va, Vb, Vc
01	01	MUL.B/F Va, Vb, Vc
01	10	MUL.H/F Va, Vb, Vc

3.4.9 MULHI

Compute the upper part of the product of two signed integer operands.

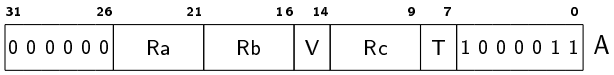


a ← (int(b) × int(c)) >> bits

V	T	Assembler
00	00	MULHI Sa, Sb, Sc
00	01	MULHI.B Sa, Sb, Sc
00	10	MULHI.H Sa, Sb, Sc
10	00	MULHI Va, Vb, Sc
10	01	MULHI.B Va, Vb, Sc
10	10	MULHI.H Va, Vb, Sc
11	00	MULHI Va, Vb, Vc
11	01	MULHI.B Va, Vb, Vc
11	10	MULHI.H Va, Vb, Vc
01	00	MULHI/F Va, Vb, Vc
01	01	MULHI.B/F Va, Vb, Vc
01	10	MULHI.H/F Va, Vb, Vc

3.4.10 MULHIU

Compute the upper part of the product of two unsigned integer operands.

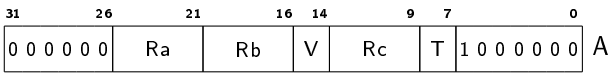


a ← (uint(b) × uint(c)) >> bits

V	T	Assembler
00	00	MULHIU Sa, Sb, Sc
00	01	MULHIU.B Sa, Sb, Sc
00	10	MULHIU.H Sa, Sb, Sc
10	00	MULHIU Va, Vb, Sc
10	01	MULHIU.B Va, Vb, Sc
10	10	MULHIU.H Va, Vb, Sc
11	00	MULHIU Va, Vb, Vc
11	01	MULHIU.B Va, Vb, Vc
11	10	MULHIU.H Va, Vb, Vc
01	00	MULHIU/F Va, Vb, Vc
01	01	MULHIU.B/F Va, Vb, Vc
01	10	MULHIU.H/F Va, Vb, Vc

3.4.11 MULQ

Compute the product of two fixed point operands.

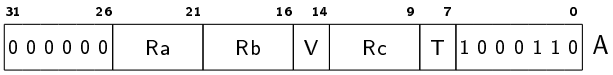


a ← (int(b) × int(c)) >> (bits-1)

V	T	Assembler
00	00	MULQ Sa, Sb, Sc
00	01	MULQ.B Sa, Sb, Sc
00	10	MULQ.H Sa, Sb, Sc
10	00	MULQ Va, Vb, Sc
10	01	MULQ.B Va, Vb, Sc
10	10	MULQ.H Va, Vb, Sc
11	00	MULQ Va, Vb, Vc
11	01	MULQ.B Va, Vb, Vc
11	10	MULQ.H Va, Vb, Vc
01	00	MULQ/F Va, Vb, Vc
01	01	MULQ.B/F Va, Vb, Vc
01	10	MULQ.H/F Va, Vb, Vc

3.4.12 REM

Compute the modulo of two signed integer operands.

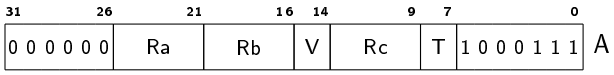


a ← int(b) % int(c)

V	T	Assembler
00	00	REM Sa, Sb, Sc
00	01	REM.B Sa, Sb, Sc
00	10	REM.H Sa, Sb, Sc
10	00	REM Va, Vb, Sc
10	01	REM.B Va, Vb, Sc
10	10	REM.H Va, Vb, Sc
11	00	REM Va, Vb, Vc
11	01	REM.B Va, Vb, Vc
11	10	REM.H Va, Vb, Vc
01	00	REM/F Va, Vb, Vc
01	01	REM.B/F Va, Vb, Vc
01	10	REM.H/F Va, Vb, Vc

3.4.13 REMU

Compute the modulo of two unsigned integer operands.

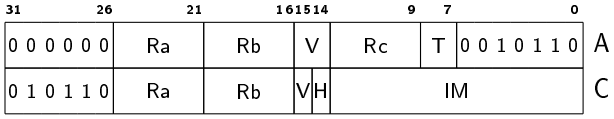



```
a ← uint(b) % uint(c)
```

V	T	Assembler
00	00	REMU Sa, Sb, Sc
00	01	REMU.B Sa, Sb, Sc
00	10	REMU.H Sa, Sb, Sc
10	00	REMU Va, Vb, Sc
10	01	REMU.B Va, Vb, Sc
10	10	REMU.H Va, Vb, Sc
11	00	REMU Va, Vb, Vc
11	01	REMU.B Va, Vb, Vc
11	10	REMU.H Va, Vb, Vc
01	00	REMU/F Va, Vb, Vc
01	01	REMU.B/F Va, Vb, Vc
01	10	REMU.H/F Va, Vb, Vc

3.4.14 SUB

Compute the difference of two integer operands.




```
a ← int(c) - int(b)
```

Fmt	V	T	Assembler
A	00	00	SUB Sa, Sc, Sb
A	00	01	SUB.B Sa, Sc, Sb
A	00	10	SUB.H Sa, Sc, Sb
A	10	00	SUB Va, Sc, Vb
A	10	01	SUB.B Va, Sc, Vb
A	10	10	SUB.H Va, Sc, Vb
A	11	00	SUB Va, Vc, Vb
A	11	01	SUB.B Va, Vc, Vb
A	11	10	SUB.H Va, Vc, Vb
A	01	00	SUB/F Va, Vc, Vb
A	01	01	SUB.B/F Va, Vc, Vb
A	01	10	SUB.H/F Va, Vc, Vb
C	00	00	SUB Sa, #ext14(H,IM), Sb
C	10	00	SUB Va, #ext14(H,IM), Vb

Note

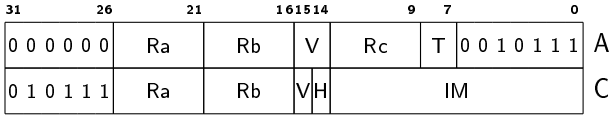
The instruction actually subtracts the first source operand from the second source operand. However, in the assembler syntax the order of the source operands is reversed compared to how the operands are encoded in the instruction word, in order to make the assembler syntax more natural.

The advantage is that it is possible to subtract a register operand from an immediate operand (subtracting an immediate operand from a register operand can be implemented with the ADD instruction, using a negated immediate operand).

3.5 Integer comparison

3.5.1 SEQ

Compare two integer operands, and set all bits of the result to 1 if the operands are equal, otherwise set all bits of the result to 0.

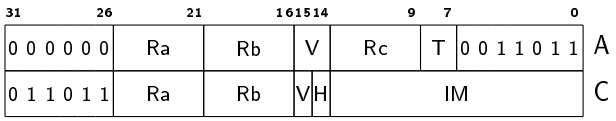



```
if b = c then
  a ← ones(bits)
else
  a ← zeros(bits)
```

Fmt	V	T	Assembler
A	00	00	SEQ Sa, Sb, Sc
A	00	01	SEQ.B Sa, Sb, Sc
A	00	10	SEQ.H Sa, Sb, Sc
A	10	00	SEQ Va, Vb, Sc
A	10	01	SEQ.B Va, Vb, Sc
A	10	10	SEQ.H Va, Vb, Sc
A	11	00	SEQ Va, Vb, Vc
A	11	01	SEQ.B Va, Vb, Vc
A	11	10	SEQ.H Va, Vb, Vc
A	01	00	SEQ/F Va, Vb, Vc
A	01	01	SEQ.B/F Va, Vb, Vc
A	01	10	SEQ.H/F Va, Vb, Vc
C	00	00	SEQ Sa, Sb, #ext14(H,IM)
C	10	00	SEQ Va, Vb, #ext14(H,IM)

3.5.2 SLE

Compare two signed integer operands, and set all bits of the result to 1 if the first operand is less than or equal to the second operand, otherwise set all bits of the result to 0.

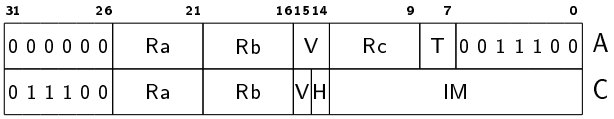



```
if int(b) ≤ int(c) then
  a ← ones(bits)
else
  a ← zeros(bits)
```

Fmt	V	T	Assembler
A	00	00	SLE Sa, Sb, Sc
A	00	01	SLE.B Sa, Sb, Sc
A	00	10	SLE.H Sa, Sb, Sc
A	10	00	SLE Va, Vb, Sc
A	10	01	SLE.B Va, Vb, Sc
A	10	10	SLE.H Va, Vb, Sc
A	11	00	SLE Va, Vb, Vc
A	11	01	SLE.B Va, Vb, Vc
A	11	10	SLE.H Va, Vb, Vc
A	01	00	SLE/F Va, Vb, Vc
A	01	01	SLE.B/F Va, Vb, Vc
A	01	10	SLE.H/F Va, Vb, Vc
C	00	00	SLE Sa, Sb, #ext14(H,IM)
C	10	00	SLE Va, Vb, #ext14(H,IM)

3.5.3 SLEU

Compare two unsigned integer operands, and set all bits of the result to 1 if the first operand is less than or equal to the second operand, otherwise set all bits of the result to 0.

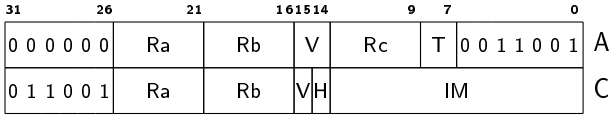



```
if uint(b) ≤ uint(c) then
  a ← ones(bits)
else
  a ← zeros(bits)
```

Fmt	V	T	Assembler
A	00	00	SLEU Sa, Sb, Sc
A	00	01	SLEU.B Sa, Sb, Sc
A	00	10	SLEU.H Sa, Sb, Sc
A	10	00	SLEU Va, Vb, Sc
A	10	01	SLEU.B Va, Vb, Sc
A	10	10	SLEU.H Va, Vb, Sc
A	11	00	SLEU Va, Vb, Vc
A	11	01	SLEU.B Va, Vb, Vc
A	11	10	SLEU.H Va, Vb, Vc
A	01	00	SLEU/F Va, Vb, Vc
A	01	01	SLEU.B/F Va, Vb, Vc
A	01	10	SLEU.H/F Va, Vb, Vc
C	00	00	SLEU Sa, Sb, #ext14(H,IM)
C	10	00	SLEU Va, Vb, #ext14(H,IM)

3.5.4 SLT

Compare two signed integer operands, and set all bits of the result to 1 if the first operand is less than the second operand, otherwise set all bits of the result to 0.

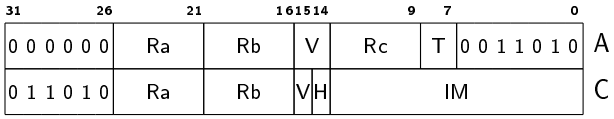



```
if int(b) < int(c) then
  a ← ones(bits)
else
  a ← zeros(bits)
```

Fmt	V	T	Assembler
A	00	00	SLT Sa, Sb, Sc
A	00	01	SLT.B Sa, Sb, Sc
A	00	10	SLT.H Sa, Sb, Sc
A	10	00	SLT Va, Vb, Sc
A	10	01	SLT.B Va, Vb, Sc
A	10	10	SLT.H Va, Vb, Sc
A	11	00	SLT Va, Vb, Vc
A	11	01	SLT.B Va, Vb, Vc
A	11	10	SLT.H Va, Vb, Vc
A	01	00	SLT/F Va, Vb, Vc
A	01	01	SLT.B/F Va, Vb, Vc
A	01	10	SLT.H/F Va, Vb, Vc
C	00	00	SLT Sa, Sb, #ext14(H,IM)
C	10	00	SLT Va, Vb, #ext14(H,IM)

3.5.5 SLTU

Compare two unsigned integer operands, and set all bits of the result to 1 if the first operand is less than the second operand, otherwise set all bits of the result to 0.

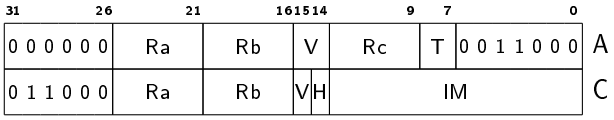



```
if uint(b) < uint(c) then
  a ← ones(bits)
else
  a ← zeros(bits)
```

Fmt	V	T	Assembler
A	00	00	SLTU Sa, Sb, Sc
A	00	01	SLTU.B Sa, Sb, Sc
A	00	10	SLTU.H Sa, Sb, Sc
A	10	00	SLTU Va, Vb, Sc
A	10	01	SLTU.B Va, Vb, Sc
A	10	10	SLTU.H Va, Vb, Sc
A	11	00	SLTU Va, Vb, Vc
A	11	01	SLTU.B Va, Vb, Vc
A	11	10	SLTU.H Va, Vb, Vc
A	01	00	SLTU/F Va, Vb, Vc
A	01	01	SLTU.B/F Va, Vb, Vc
A	01	10	SLTU.H/F Va, Vb, Vc
C	00	00	SLTU Sa, Sb, #ext14(H,IM)
C	10	00	SLTU Va, Vb, #ext14(H,IM)

3.5.6 SNE

Compare two integer operands, and set all bits of the result to 1 if the operands are not equal, otherwise set all bits of the result to 0.



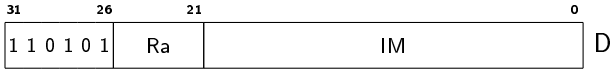

```
if b ≠ c then
  a ← ones(bits)
else
  a ← zeros(bits)
```

Fmt	V	T	Assembler
A	00	00	SNE Sa, Sb, Sc
A	00	01	SNE.B Sa, Sb, Sc
A	00	10	SNE.H Sa, Sb, Sc
A	10	00	SNE Va, Vb, Sc
A	10	01	SNE.B Va, Vb, Sc
A	10	10	SNE.H Va, Vb, Sc
A	11	00	SNE Va, Vb, Vc
A	11	01	SNE.B Va, Vb, Vc
A	11	10	SNE.H Va, Vb, Vc
A	01	00	SNE/F Va, Vb, Vc
A	01	01	SNE.B/F Va, Vb, Vc
A	01	10	SNE.H/F Va, Vb, Vc
C	00	00	SNE Sa, Sb, #ext14(H,IM)
C	10	00	SNE Va, Vb, #ext14(H,IM)

3.6 Branch

3.6.1 BGE

Branch to the PC-relative target if the first source operand is a signed integer value that is greater than or equal to zero.



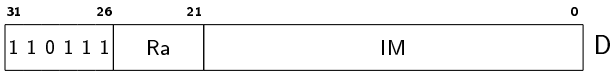

```
if int(a) ≥ 0 then
  PC ← PC + int(IM)×4
```

Assembler

```
BGE     Sa, #target
```

3.6.2 BGT

Branch to the PC-relative target if the first source operand is a signed integer value that is greater than zero.



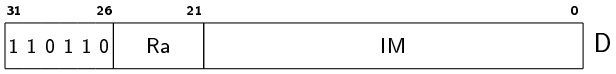

```
if int(a) > 0 then
  PC ← PC + int(IM)×4
```

Assembler

```
BGT     Sa, #target
```

3.6.3 BLE

Branch to the PC-relative target if the first source operand is a signed integer value that is less than or equal to zero.



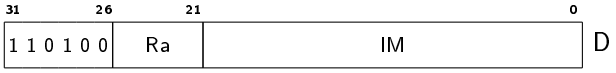

```
if int(a) ≤ 0 then
  PC ← PC + int(IM)×4
```

Assembler

```
BLE     Sa, #target
```

3.6.4 BLT

Branch to the PC-relative target if the first source operand is a signed integer value that is less than zero.



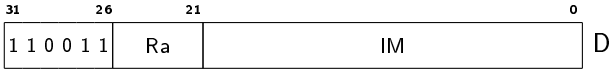

```
if int(a) < 0 then
  PC ← PC + int(IM)×4
```

Assembler

```
BLT     Sa, #target
```

3.6.5 BNS

Branch to the PC-relative target if at least one of the bits of the first source operand is zero.



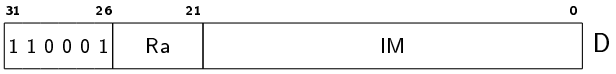

```
if a ≠ ones(32) then
  PC ← PC + int(IM)×4
```

Assembler

```
BNS     Sa, #target
```

3.6.6 BNZ

Branch to the PC-relative target if at least one of the bits of the first source operand is non-zero.



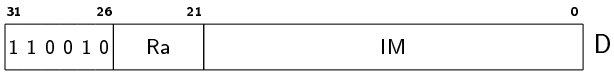

```
if a ≠ zeros(32) then
  PC ← PC + int(IM)×4
```

Assembler

```
BNZ     Sa, #target
```

3.6.7 BS

Branch to the PC-relative target if all bits of the first source operand are non-zero.



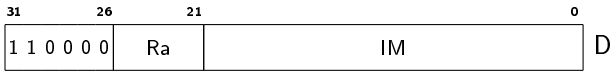

```
if a = ones(32) then
  PC ← PC + int(IM)×4
```

Assembler

```
BS      Sa, #target
```

3.6.8 BZ

Branch to the PC-relative target if all bits of the first source operand are zero.



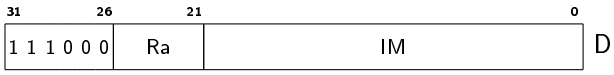

```
if a = zeros(32) then
  PC ← PC + int(IM)×4
```

Assembler

```
BZ      Sa, #target
```

3.6.9 J

Jump to the target address that is formed by computing the sum of the register operand and the sign-extended immediate value multiplied by four.




```
PC ← int(a) + int(IM)×4
```

Assembler

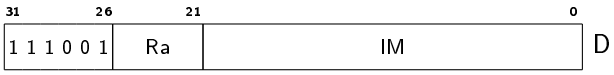
```
J       Sa, #target
```

> **Note**
>
> If the register operand is PC, a PC-relative branch with an effective range of ±4MiB is performed. To extend the range to the full address space, use J in combination with a preceding ADDPCHI.
>
> If the register operand is Z, an absolute branch is performed. Possible target addresses are 0x00000000 to 0x003FFFFC and 0xFFC00000 to 0xFFFFFFFC. To extend the range to the full address space, use J in combination with a preceding LDHI.

> If the register operand is LR and the immediate value is zero, the operation will return the program flow to the caller (RET is an alias for J LR, #0).

3.6.10 JL

Jump and link. The current value of PC plus four is stored in the LR register, and the new PC is set to the target address that is formed by computing the sum of the register operand and the sign-extended immediate value multiplied by four.




```
adr ← int(a) + int(IM)×4
LR ← int(PC) + 4
PC ← adr
```

Assembler

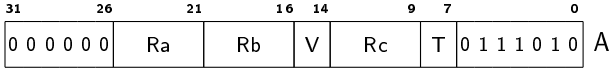
```
JL      Sa, #target
```

> **Note**
>
> If the register operand is PC, a PC-relative branch with an effective range of ±4MiB is performed. To extend the range to the full address space, use JL in combination with a preceding ADDPCHI.
>
> If the register operand is Z, an absolute branch is performed. Possible target addresses are 0x00000000 to 0x003FFFFC and 0xFFC00000 to 0xFFFFFFFC. To extend the range to the full address space, use JL in combination with a preceding LDHI.

3.7 Saturating and halving arithmetic

3.7.1 ADDH

Compute the half sum of two signed integer operands.

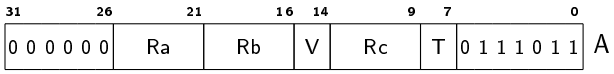



```
a ← (int(b) + int(c)) >> 1
```

V	T	Assembler
00	00	ADDH Sa, Sb, Sc
00	01	ADDH.B Sa, Sb, Sc
00	10	ADDH.H Sa, Sb, Sc
10	00	ADDH Va, Vb, Sc
10	01	ADDH.B Va, Vb, Sc
10	10	ADDH.H Va, Vb, Sc
11	00	ADDH Va, Vb, Vc
11	01	ADDH.B Va, Vb, Vc
11	10	ADDH.H Va, Vb, Vc
01	00	ADDH/F Va, Vb, Vc
01	01	ADDH.B/F Va, Vb, Vc
01	10	ADDH.H/F Va, Vb, Vc

3.7.2 ADDHU

Compute the half sum of two unsigned integer operands.

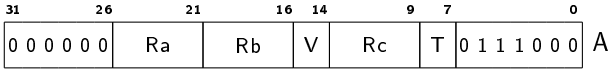



```
a ← (uint(b) + uint(c)) >> 1
```

V	T	Assembler
00	00	ADDHU Sa, Sb, Sc
00	01	ADDHU.B Sa, Sb, Sc
00	10	ADDHU.H Sa, Sb, Sc
10	00	ADDHU Va, Vb, Sc
10	01	ADDHU.B Va, Vb, Sc
10	10	ADDHU.H Va, Vb, Sc
11	00	ADDHU Va, Vb, Vc
11	01	ADDHU.B Va, Vb, Vc
11	10	ADDHU.H Va, Vb, Vc
01	00	ADDHU/F Va, Vb, Vc
01	01	ADDHU.B/F Va, Vb, Vc
01	10	ADDHU.H/F Va, Vb, Vc

3.7.3 ADDS

Compute the saturated sum of two signed integer operands.

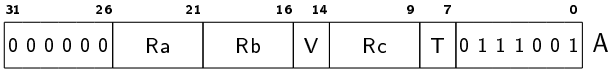



```
a ← sat(int(b) + int(c), bits)
```

V	T	Assembler
00	00	ADDS Sa, Sb, Sc
00	01	ADDS.B Sa, Sb, Sc
00	10	ADDS.H Sa, Sb, Sc
10	00	ADDS Va, Vb, Sc
10	01	ADDS.B Va, Vb, Sc
10	10	ADDS.H Va, Vb, Sc
11	00	ADDS Va, Vb, Vc
11	01	ADDS.B Va, Vb, Vc
11	10	ADDS.H Va, Vb, Vc
01	00	ADDS/F Va, Vb, Vc
01	01	ADDS.B/F Va, Vb, Vc
01	10	ADDS.H/F Va, Vb, Vc

3.7.4 ADDSU

Compute the saturated sum of two unsigned integer operands.

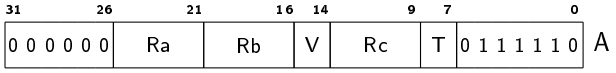



```
a ← satu(uint(b) + uint(c), bits)
```

V	T	Assembler
00	00	ADDSU Sa, Sb, Sc
00	01	ADDSU.B Sa, Sb, Sc
00	10	ADDSU.H Sa, Sb, Sc
10	00	ADDSU Va, Vb, Sc
10	01	ADDSU.B Va, Vb, Sc
10	10	ADDSU.H Va, Vb, Sc
11	00	ADDSU Va, Vb, Vc
11	01	ADDSU.B Va, Vb, Vc
11	10	ADDSU.H Va, Vb, Vc
01	00	ADDSU/F Va, Vb, Vc
01	01	ADDSU.B/F Va, Vb, Vc
01	10	ADDSU.H/F Va, Vb, Vc

3.7.5 SUBH

Compute the half difference of two signed integer operands.

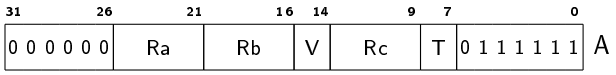



```
a ← (int(b) - int(c)) >> 1
```

V	T	Assembler
00	00	SUBH Sa, Sb, Sc
00	01	SUBH.B Sa, Sb, Sc
00	10	SUBH.H Sa, Sb, Sc
10	00	SUBH Va, Vb, Sc
10	01	SUBH.B Va, Vb, Sc
10	10	SUBH.H Va, Vb, Sc
11	00	SUBH Va, Vb, Vc
11	01	SUBH.B Va, Vb, Vc
11	10	SUBH.H Va, Vb, Vc
01	00	SUBH/F Va, Vb, Vc
01	01	SUBH.B/F Va, Vb, Vc
01	10	SUBH.H/F Va, Vb, Vc

3.7.6 SUBHU

Compute the half difference of two unsigned integer operands.

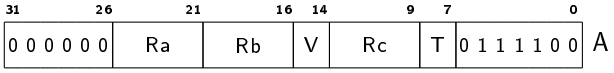



```
a ← (uint(b) - uint(c)) >> 1
```

V	T	Assembler
00	00	SUBHU Sa, Sb, Sc
00	01	SUBHU.B Sa, Sb, Sc
00	10	SUBHU.H Sa, Sb, Sc
10	00	SUBHU Va, Vb, Sc
10	01	SUBHU.B Va, Vb, Sc
10	10	SUBHU.H Va, Vb, Sc
11	00	SUBHU Va, Vb, Vc
11	01	SUBHU.B Va, Vb, Vc
11	10	SUBHU.H Va, Vb, Vc
01	00	SUBHU/F Va, Vb, Vc
01	01	SUBHU.B/F Va, Vb, Vc
01	10	SUBHU.H/F Va, Vb, Vc

3.7.7 SUBS

Compute the saturated difference of two signed integer operands.

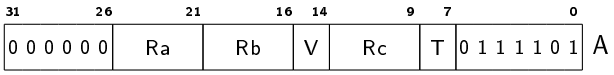



```
a ← sat(int(b) - int(c), bits)
```

V	T	Assembler
00	00	SUBS Sa, Sb, Sc
00	01	SUBS.B Sa, Sb, Sc
00	10	SUBS.H Sa, Sb, Sc
10	00	SUBS Va, Vb, Sc
10	01	SUBS.B Va, Vb, Sc
10	10	SUBS.H Va, Vb, Sc
11	00	SUBS Va, Vb, Vc
11	01	SUBS.B Va, Vb, Vc
11	10	SUBS.H Va, Vb, Vc
01	00	SUBS/F Va, Vb, Vc
01	01	SUBS.B/F Va, Vb, Vc
01	10	SUBS.H/F Va, Vb, Vc

3.7.8 SUBSU

Compute the saturated difference of two unsigned integer operands.



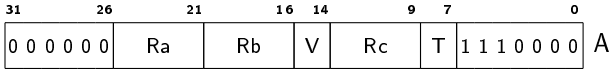

```
a ← satu(uint(b) - uint(c), bits)
```

V	T	Assembler
00	00	SUBSU Sa, Sb, Sc
00	01	SUBSU.B Sa, Sb, Sc
00	10	SUBSU.H Sa, Sb, Sc
10	00	SUBSU Va, Vb, Sc
10	01	SUBSU.B Va, Vb, Sc
10	10	SUBSU.H Va, Vb, Sc
11	00	SUBSU Va, Vb, Vc
11	01	SUBSU.B Va, Vb, Vc
11	10	SUBSU.H Va, Vb, Vc
01	00	SUBSU/F Va, Vb, Vc
01	01	SUBSU.B/F Va, Vb, Vc
01	10	SUBSU.H/F Va, Vb, Vc

3.8 Floating-point arithmetic

3.8.1 FADD

Compute the sum of two floating-point operands.

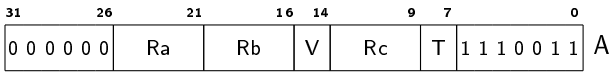



```
a ← float(b) + float(c)
```

V	T	Assembler
00	00	FADD Sa, Sb, Sc
00	01	FADD.B Sa, Sb, Sc
00	10	FADD.H Sa, Sb, Sc
10	00	FADD Va, Vb, Sc
10	01	FADD.B Va, Vb, Sc
10	10	FADD.H Va, Vb, Sc
11	00	FADD Va, Vb, Vc
11	01	FADD.B Va, Vb, Vc
11	10	FADD.H Va, Vb, Vc
01	00	FADD/F Va, Vb, Vc
01	01	FADD.B/F Va, Vb, Vc
01	10	FADD.H/F Va, Vb, Vc

3.8.2 FDIV

Compute the quotient of two floating-point operands.

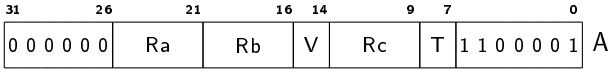



```
a ← float(b) / float(c)
```

V	T	Assembler
00	00	FDIV Sa, Sb, Sc
00	01	FDIV.B Sa, Sb, Sc
00	10	FDIV.H Sa, Sb, Sc
10	00	FDIV Va, Vb, Sc
10	01	FDIV.B Va, Vb, Sc
10	10	FDIV.H Va, Vb, Sc
11	00	FDIV Va, Vb, Vc
11	01	FDIV.B Va, Vb, Vc
11	10	FDIV.H Va, Vb, Vc
01	00	FDIV/F Va, Vb, Vc
01	01	FDIV.B/F Va, Vb, Vc
01	10	FDIV.H/F Va, Vb, Vc

3.8.3 FMAX

Return the maximum value of two floating-point operands.

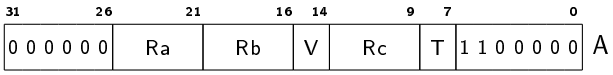



```
a ← max(float(b), float(c))
```

V	T	Assembler
00	00	FMAX Sa, Sb, Sc
00	01	FMAX.B Sa, Sb, Sc
00	10	FMAX.H Sa, Sb, Sc
10	00	FMAX Va, Vb, Sc
10	01	FMAX.B Va, Vb, Sc
10	10	FMAX.H Va, Vb, Sc
11	00	FMAX Va, Vb, Vc
11	01	FMAX.B Va, Vb, Vc
11	10	FMAX.H Va, Vb, Vc
01	00	FMAX/F Va, Vb, Vc
01	01	FMAX.B/F Va, Vb, Vc
01	10	FMAX.H/F Va, Vb, Vc

3.8.4 FMIN

Return the minimum value of two floating-point operands.

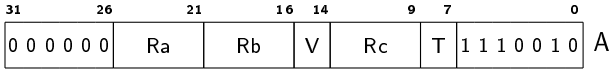



```
a ← min(float(b), float(c))
```

V	T	Assembler
00	00	FMIN Sa, Sb, Sc
00	01	FMIN.B Sa, Sb, Sc
00	10	FMIN.H Sa, Sb, Sc
10	00	FMIN Va, Vb, Sc
10	01	FMIN.B Va, Vb, Sc
10	10	FMIN.H Va, Vb, Sc
11	00	FMIN Va, Vb, Vc
11	01	FMIN.B Va, Vb, Vc
11	10	FMIN.H Va, Vb, Vc
01	00	FMIN/F Va, Vb, Vc
01	01	FMIN.B/F Va, Vb, Vc
01	10	FMIN.H/F Va, Vb, Vc

3.8.5 FMUL

Compute the product of two floating-point operands.

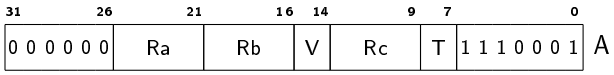



```
a ← float(b) × float(c)
```

V	T	Assembler
00	00	FMUL Sa, Sb, Sc
00	01	FMUL.B Sa, Sb, Sc
00	10	FMUL.H Sa, Sb, Sc
10	00	FMUL Va, Vb, Sc
10	01	FMUL.B Va, Vb, Sc
10	10	FMUL.H Va, Vb, Sc
11	00	FMUL Va, Vb, Vc
11	01	FMUL.B Va, Vb, Vc
11	10	FMUL.H Va, Vb, Vc
01	00	FMUL/F Va, Vb, Vc
01	01	FMUL.B/F Va, Vb, Vc
01	10	FMUL.H/F Va, Vb, Vc

3.8.6 FSUB

Compute the difference of two floating-point operands.



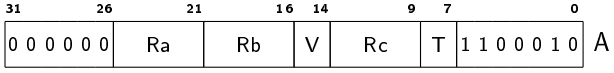

```
a ← float(b) - float(c)
```

V	T	Assembler
00	00	FSUB Sa, Sb, Sc
00	01	FSUB.B Sa, Sb, Sc
00	10	FSUB.H Sa, Sb, Sc
10	00	FSUB Va, Vb, Sc
10	01	FSUB.B Va, Vb, Sc
10	10	FSUB.H Va, Vb, Sc
11	00	FSUB Va, Vb, Vc
11	01	FSUB.B Va, Vb, Vc
11	10	FSUB.H Va, Vb, Vc
01	00	FSUB/F Va, Vb, Vc
01	01	FSUB.B/F Va, Vb, Vc
01	10	FSUB.H/F Va, Vb, Vc

3.9 Floating-point comparison

3.9.1 FSEQ

Compare two floating-point operands, and set all bits of the result to 1 if the first operand is equal to the second operand, otherwise set all bits of the result to 0.

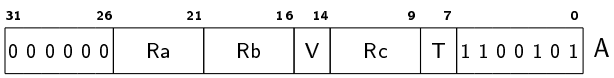



```
if float(b) = float(c) then
  a ← ones(bits)
else
  a ← zeros(bits)
```

V	T	Assembler
00	00	FSEQ Sa, Sb, Sc
00	01	FSEQ.B Sa, Sb, Sc
00	10	FSEQ.H Sa, Sb, Sc
10	00	FSEQ Va, Vb, Sc
10	01	FSEQ.B Va, Vb, Sc
10	10	FSEQ.H Va, Vb, Sc
11	00	FSEQ Va, Vb, Vc
11	01	FSEQ.B Va, Vb, Vc
11	10	FSEQ.H Va, Vb, Vc
01	00	FSEQ/F Va, Vb, Vc
01	01	FSEQ.B/F Va, Vb, Vc
01	10	FSEQ.H/F Va, Vb, Vc

3.9.2 FSLE

Compare two floating-point operands, and set all bits of the result to 1 if the first operand is less than or equal to the second operand, otherwise set all bits of the result to 0.

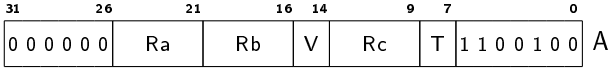



```
if float(b) ≤ float(c) then
  a ← ones(bits)
else
  a ← zeros(bits)
```

V	T	Assembler
00	00	FSLE Sa, Sb, Sc
00	01	FSLE.B Sa, Sb, Sc
00	10	FSLE.H Sa, Sb, Sc
10	00	FSLE Va, Vb, Sc
10	01	FSLE.B Va, Vb, Sc
10	10	FSLE.H Va, Vb, Sc
11	00	FSLE Va, Vb, Vc
11	01	FSLE.B Va, Vb, Vc
11	10	FSLE.H Va, Vb, Vc
01	00	FSLE/F Va, Vb, Vc
01	01	FSLE.B/F Va, Vb, Vc
01	10	FSLE.H/F Va, Vb, Vc

3.9.3 FSLT

Compare two floating-point operands, and set all bits of the result to 1 if the first operand is less than the second operand, otherwise set all bits of the result to 0.

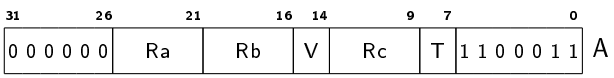



```
if float(b) < float(c) then
  a ← ones(bits)
else
  a ← zeros(bits)
```

V	T	Assembler
00	00	FSLT Sa, Sb, Sc
00	01	FSLT.B Sa, Sb, Sc
00	10	FSLT.H Sa, Sb, Sc
10	00	FSLT Va, Vb, Sc
10	01	FSLT.B Va, Vb, Sc
10	10	FSLT.H Va, Vb, Sc
11	00	FSLT Va, Vb, Vc
11	01	FSLT.B Va, Vb, Vc
11	10	FSLT.H Va, Vb, Vc
01	00	FSLT/F Va, Vb, Vc
01	01	FSLT.B/F Va, Vb, Vc
01	10	FSLT.H/F Va, Vb, Vc

3.9.4 FSNE

Compare two floating-point operands, and set all bits of the result to 1 if the first operand is not equal to the second operand, otherwise set all bits of the result to 0.

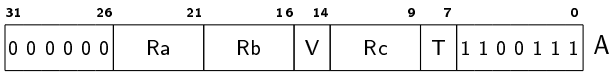


```
if float(b) ≠ float(c) then
  a ← ones(bits)
else
  a ← zeros(bits)
```

V	T	Assembler
00	00	FSNE Sa, Sb, Sc
00	01	FSNE.B Sa, Sb, Sc
00	10	FSNE.H Sa, Sb, Sc
10	00	FSNE Va, Vb, Sc
10	01	FSNE.B Va, Vb, Sc
10	10	FSNE.H Va, Vb, Sc
11	00	FSNE Va, Vb, Vc
11	01	FSNE.B Va, Vb, Vc
11	10	FSNE.H Va, Vb, Vc
01	00	FSNE/F Va, Vb, Vc
01	01	FSNE.B/F Va, Vb, Vc
01	10	FSNE.H/F Va, Vb, Vc

3.9.5 FSORD

Set all bits of the result to 1 if both of the source operands are ordered (i.e. non-NaN), otherwise set all bits of the result to 0.

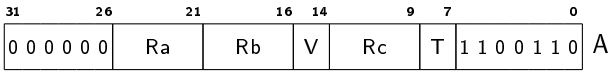



```
if ¬isnan(b) ∨ ¬isnan(c) then
  a ← ones(bits)
else
  a ← zeros(bits)
```

V	T	Assembler
00	00	FSORD Sa, Sb, Sc
00	01	FSORD.B Sa, Sb, Sc
00	10	FSORD.H Sa, Sb, Sc
10	00	FSORD Va, Vb, Sc
10	01	FSORD.B Va, Vb, Sc
10	10	FSORD.H Va, Vb, Sc
11	00	FSORD Va, Vb, Vc
11	01	FSORD.B Va, Vb, Vc
11	10	FSORD.H Va, Vb, Vc
01	00	FSORD/F Va, Vb, Vc
01	01	FSORD.B/F Va, Vb, Vc
01	10	FSORD.H/F Va, Vb, Vc

3.9.6 FSUNORD

Set all bits of the result to 1 if any of the source operands are undordered (i.e. NaN), otherwise set all bits of the result to 0.




```
if isnan(b) ∧ isnan(c) then
  a ← ones(bits)
else
  a ← zeros(bits)
```

V	T	Assembler
00	00	FSUNORD Sa, Sb, Sc
00	01	FSUNORD.B Sa, Sb, Sc
00	10	FSUNORD.H Sa, Sb, Sc
10	00	FSUNORD Va, Vb, Sc
10	01	FSUNORD.B Va, Vb, Sc
10	10	FSUNORD.H Va, Vb, Sc
11	00	FSUNORD Va, Vb, Vc
11	01	FSUNORD.B Va, Vb, Vc
11	10	FSUNORD.H Va, Vb, Vc
01	00	FSUNORD/F Va, Vb, Vc
01	01	FSUNORD.B/F Va, Vb, Vc
01	10	FSUNORD.H/F Va, Vb, Vc

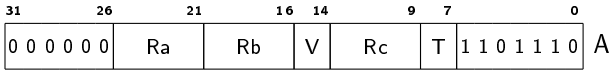
3.10 Floating-point conversion

3.10.1 FPACK

Convert and pack two floating-point operands into a single operand.

The precision of the two source operands are halved. The first source operand is packed and stored in the upper half of the destination operand, and the second source operand is packed and stored in the lower half of the destination operand.

> **TODO**
>
> *Define pseudocode.*
>
> *The packed byte (.B) variant is undefined and should not be listed.*

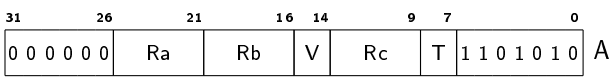


V	T	Assembler
00	00	FPACK Sa, Sb, Sc
00	01	FPACK.B Sa, Sb, Sc
00	10	FPACK.H Sa, Sb, Sc
10	00	FPACK Va, Vb, Sc
10	01	FPACK.B Va, Vb, Sc
10	10	FPACK.H Va, Vb, Sc
11	00	FPACK Va, Vb, Vc
11	01	FPACK.B Va, Vb, Vc
11	10	FPACK.H Va, Vb, Vc
01	00	FPACK/F Va, Vb, Vc
01	01	FPACK.B/F Va, Vb, Vc
01	10	FPACK.H/F Va, Vb, Vc

3.10.2 FTOI

Convert a floating-point value to a signed integer value, without rounding. The integer offset provided by the second source operand is added to the floating-point exponent before the conversion.

> **TODO**
>
> *Define pseudocode.*

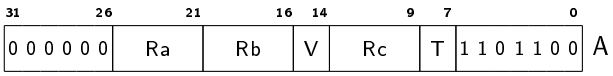


V	T	Assembler
00	00	FTOI Sa, Sb, Sc
00	01	FTOI.B Sa, Sb, Sc
00	10	FTOI.H Sa, Sb, Sc
10	00	FTOI Va, Vb, Sc
10	01	FTOI.B Va, Vb, Sc
10	10	FTOI.H Va, Vb, Sc
11	00	FTOI Va, Vb, Vc
11	01	FTOI.B Va, Vb, Vc
11	10	FTOI.H Va, Vb, Vc
01	00	FTOI/F Va, Vb, Vc
01	01	FTOI.B/F Va, Vb, Vc
01	10	FTOI.H/F Va, Vb, Vc

3.10.3 FTOIR

Convert a floating-point value to a signed integer value, with rounding. The integer offset provided by the second source operand is added to the floating-point exponent before the conversion.

> **TODO**
>
> *Define pseudocode.*



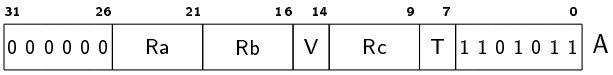
V	T	Assembler
00	00	FTOIR Sa, Sb, Sc
00	01	FTOIR.B Sa, Sb, Sc
00	10	FTOIR.H Sa, Sb, Sc
10	00	FTOIR Va, Vb, Sc
10	01	FTOIR.B Va, Vb, Sc
10	10	FTOIR.H Va, Vb, Sc
11	00	FTOIR Va, Vb, Vc
11	01	FTOIR.B Va, Vb, Vc
11	10	FTOIR.H Va, Vb, Vc
01	00	FTOIR/F Va, Vb, Vc
01	01	FTOIR.B/F Va, Vb, Vc
01	10	FTOIR.H/F Va, Vb, Vc

3.10.4 FTOU

Convert a floating-point value to an unsigned integer value, without rounding. The integer offset provided by the second source operand is added to the floating-point exponent before the conversion.

TODO

Define pseudocode.



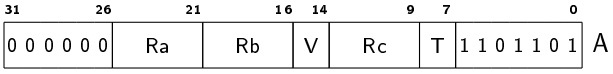
V	T	Assembler
00	00	FTOU Sa, Sb, Sc
00	01	FTOU.B Sa, Sb, Sc
00	10	FTOU.H Sa, Sb, Sc
10	00	FTOU Va, Vb, Sc
10	01	FTOU.B Va, Vb, Sc
10	10	FTOU.H Va, Vb, Sc
11	00	FTOU Va, Vb, Vc
11	01	FTOU.B Va, Vb, Vc
11	10	FTOU.H Va, Vb, Vc
01	00	FTOU/F Va, Vb, Vc
01	01	FTOU.B/F Va, Vb, Vc
01	10	FTOU.H/F Va, Vb, Vc

3.10.5 FTOUR

Convert a floating-point value to an unsigned integer value, with rounding. The integer offset provided by the second source operand is added to the floating-point exponent before the conversion.

TODO

Define pseudocode.



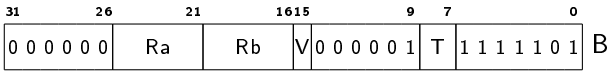
V	T	Assembler
00	00	FTOUR Sa, Sb, Sc
00	01	FTOUR.B Sa, Sb, Sc
00	10	FTOUR.H Sa, Sb, Sc
10	00	FTOUR Va, Vb, Sc
10	01	FTOUR.B Va, Vb, Sc
10	10	FTOUR.H Va, Vb, Sc
11	00	FTOUR Va, Vb, Vc
11	01	FTOUR.B Va, Vb, Vc
11	10	FTOUR.H Va, Vb, Vc
01	00	FTOUR/F Va, Vb, Vc
01	01	FTOUR.B/F Va, Vb, Vc
01	10	FTOUR.H/F Va, Vb, Vc

3.10.6 FUNPH

Unpack the high half of a packed floating-point pair. The preicison of the unpacked source floating-point value is doubled and stored in the destination operand.

TODO

Define pseudocode.



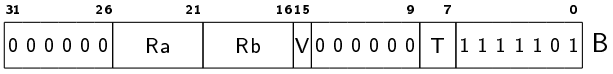
V	T	Assembler
00	00	FUNPH Sa, Sb
00	01	FUNPH.B Sa, Sb
00	10	FUNPH.H Sa, Sb
10	00	FUNPH Va, Vb
10	01	FUNPH.B Va, Vb
10	10	FUNPH.H Va, Vb

3.10.7 FUNPL

Unpack the low half of a packed floating-point pair. The preicison of the unpacked source floating-point value is doubled and stored in the destination operand.

TODO

Define pseudocode.



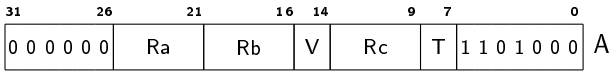
V	T	Assembler
00	00	FUNPL Sa, Sb
00	01	FUNPL.B Sa, Sb
00	10	FUNPL.H Sa, Sb
10	00	FUNPL Va, Vb
10	01	FUNPL.B Va, Vb
10	10	FUNPL.H Va, Vb

3.10.8 ITOF

Convert a signed integer value to a floating-point value. The exponent of the resulting floating-point value is subtracted by the integer offset provided by the second source operand before storing the final floating-point value in the destination operand.

TODO

Define pseudocode.



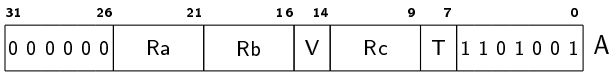
V	T	Assembler
00	00	`ITOF Sa, Sb, Sc`
00	01	`ITOF.B Sa, Sb, Sc`
00	10	`ITOF.H Sa, Sb, Sc`
10	00	`ITOF Va, Vb, Sc`
10	01	`ITOF.B Va, Vb, Sc`
10	10	`ITOF.H Va, Vb, Sc`
11	00	`ITOF Va, Vb, Vc`
11	01	`ITOF.B Va, Vb, Vc`
11	10	`ITOF.H Va, Vb, Vc`
01	00	`ITOF/F Va, Vb, Vc`
01	01	`ITOF.B/F Va, Vb, Vc`
01	10	`ITOF.H/F Va, Vb, Vc`

3.10.9 UTOF

Convert an unsigned integer value to a floating-point value. The exponent of the resulting floating-point value is subtracted by the integer offset provided by the second source operand before storing the final floating-point value in the destination operand.

TODO

Define pseudocode.

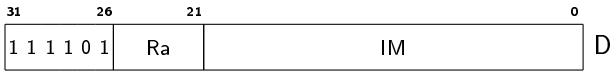


V	T	Assembler
00	00	`UTOF Sa, Sb, Sc`
00	01	`UTOF.B Sa, Sb, Sc`
00	10	`UTOF.H Sa, Sb, Sc`
10	00	`UTOF Va, Vb, Sc`
10	01	`UTOF.B Va, Vb, Sc`
10	10	`UTOF.H Va, Vb, Sc`
11	00	`UTOF Va, Vb, Vc`
11	01	`UTOF.B Va, Vb, Vc`
11	10	`UTOF.H Va, Vb, Vc`
01	00	`UTOF/F Va, Vb, Vc`
01	01	`UTOF.B/F Va, Vb, Vc`
01	10	`UTOF.H/F Va, Vb, Vc`

3.11 Miscellaneous

3.11.1 ADDPCHI

Compute the sum of the current PC and and immediate operand that is shifted to the left 11 steps before the addition.



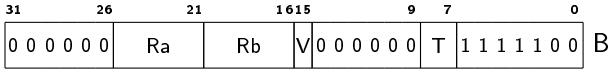

```
a ← int(PC) + (int(IM) << 11)
```

Assembler
`ADDPCHI Sa, #target@pchi`

> **Note**
>
> This instruction can be used in combination with several instructions that take an immediate operand in order to form a full 32-bit PC-relative offset. Examples of such instructions are ADD, LDW and JL.

3.11.2 CLZ

Count the number of leading zero bits in the source operand.



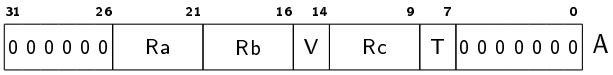

```
a ← 0
for k in bits-1 downto 0
  if b<k> = 1 then
    break
  a ← int(a) + 1
```

V	T	Assembler
00	00	`CLZ Sa, Sb`
00	01	`CLZ.B Sa, Sb`
00	10	`CLZ.H Sa, Sb`
10	00	`CLZ Va, Vb`
10	01	`CLZ.B Va, Vb`
10	10	`CLZ.H Va, Vb`

3.11.3 CPUID

Query CPU properties.

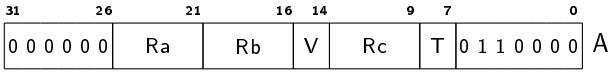
> **TODO**
>
> *Define pseudocode.*



V	Assembler
00	`CPUID Sa, Sb, Sc`
10	`CPUID Va, Vb, Sc`
11	`CPUID Va, Vb, Vc`
01	`CPUID/F Va, Vb, Vc`

3.11.4 PACK

Pack the lower parts of two integer operands.

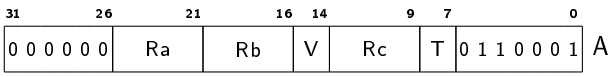



```
a ← (b << (bits/2)) | (c & ones(bits/2))
```

V	T	Assembler
00	00	`PACK Sa, Sb, Sc`
00	01	`PACK.B Sa, Sb, Sc`
00	10	`PACK.H Sa, Sb, Sc`
10	00	`PACK Va, Vb, Sc`
10	01	`PACK.B Va, Vb, Sc`
10	10	`PACK.H Va, Vb, Sc`
11	00	`PACK Va, Vb, Vc`
11	01	`PACK.B Va, Vb, Vc`
11	10	`PACK.H Va, Vb, Vc`
01	00	`PACK/F Va, Vb, Vc`
01	01	`PACK.B/F Va, Vb, Vc`
01	10	`PACK.H/F Va, Vb, Vc`

3.11.5 PACKS

Saturate and pack the lower parts of two signed integer operands.

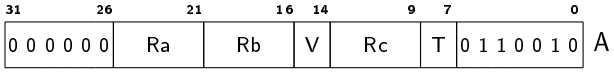



```
hi ← sat(int(b), bits/2) & ones(bits/2)
lo ← sat(int(c), bits/2) & ones(bits/2)
a ← (hi << (bits/2)) | lo
```

V	T	Assembler
00	00	PACKS Sa, Sb, Sc
00	01	PACKS.B Sa, Sb, Sc
00	10	PACKS.H Sa, Sb, Sc
10	00	PACKS Va, Vb, Sc
10	01	PACKS.B Va, Vb, Sc
10	10	PACKS.H Va, Vb, Sc
11	00	PACKS Va, Vb, Vc
11	01	PACKS.B Va, Vb, Vc
11	10	PACKS.H Va, Vb, Vc
01	00	PACKS/F Va, Vb, Vc
01	01	PACKS.B/F Va, Vb, Vc
01	10	PACKS.H/F Va, Vb, Vc

3.11.6 PACKSU

Saturate and pack the lower parts of two unsigned integer operands.

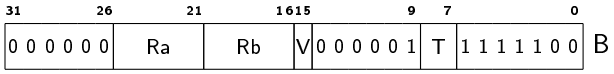



```
hi ← satu(uint(b), bits/2)
lo ← satu(uint(c), bits/2)
a ← (hi << (bits/2)) | lo
```

V	T	Assembler
00	00	PACKSU Sa, Sb, Sc
00	01	PACKSU.B Sa, Sb, Sc
00	10	PACKSU.H Sa, Sb, Sc
10	00	PACKSU Va, Vb, Sc
10	01	PACKSU.B Va, Vb, Sc
10	10	PACKSU.H Va, Vb, Sc
11	00	PACKSU Va, Vb, Vc
11	01	PACKSU.B Va, Vb, Vc
11	10	PACKSU.H Va, Vb, Vc
01	00	PACKSU/F Va, Vb, Vc
01	01	PACKSU.B/F Va, Vb, Vc
01	10	PACKSU.H/F Va, Vb, Vc

3.11.7 POPCNT

Count the number of non-zero bits in the source operand.

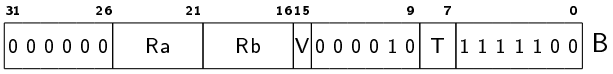



```
a ← 0
for k in 0 to bits-1
  if b<k> = 1 then
    a ← int(a) + 1
```

V	T	Assembler
00	00	POPCNT Sa, Sb
00	01	POPCNT.B Sa, Sb
00	10	POPCNT.H Sa, Sb
10	00	POPCNT Va, Vb
10	01	POPCNT.B Va, Vb
10	10	POPCNT.H Va, Vb

3.11.8 REV

Reverse the bits of the source operand.



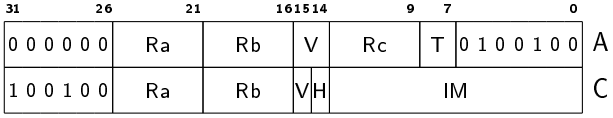

```
for k in 0 to bits-1
  a<bits-1-k> ← b<k>
```

V	T	Assembler
00	00	REV Sa, Sb
00	01	REV.B Sa, Sb
00	10	REV.H Sa, Sb
10	00	REV Va, Vb
10	01	REV.B Va, Vb
10	10	REV.H Va, Vb

3.11.9 SHUF

Shuffle bytes.

> **TODO**
>
> *Define pseudocode.*



Fmt	V	Assembler
A	00	SHUF Sa, Sb, Sc
A	10	SHUF Va, Vb, Sc
A	11	SHUF Va, Vb, Vc
A	01	SHUF/F Va, Vb, Vc
C	00	SHUF Sa, Sb, #ext14(H,IM)
C	10	SHUF Va, Vb, #ext14(H,IM)

Appendix A

Examples

This is a non-normative section that contains programs that exemplify various aspects of the MRISC32 instruction set architecture.

A.1 Vector operation

A.1.1 saxpy

```
; void saxpy(size_t n, const float a, const float *x, float *y)
; {
;   for (size_t i = 0; i < n; i++)
;     y[i] = a * x[i] + y[i];
; }
;
; Register arguments:
;   s1 - n
;   s2 - a
;   s3 - x
;   s4 - y

saxpy:
    bz    s1, 2f         ; Nothing to do?
    cpuid s5, z, z       ; Query the maximum vector length
1:
    minu  vl, s5, s1     ; Define the operation vector length
    sub   s1, s1, vl     ; Decrement loop counter
    ldw   v1, s3, #4     ; Load x (element stride = 4 bytes)
    ldw   v2, s4, #4     ; Load y
    fmul  v1, v1, s2     ; x * a
    fadd  v1, v1, v2     ; + y
    stw   v1, s4, #4     ; Store y
    ldea  s3, s3, vl*4   ; Increment array pointers
    ldea  s4, s4, vl*4   ; Increment array pointers
    bnz   s1, 1b
2:
    ret
```

A.1.2 Linear interpolation

Linear interpolation can be implemented using vector gather load. Here is an example of one-dimensional floating-point interpolation.

```
; void lerp(size_t n, const float t0, const float dt, const float *x, float *y)
; {
;   float t = t0;
;   for (size_t i = 0; i < n; i++)
;   {
;     int k = (int)t;
;     float w = t - (float)k;
;     y[i] = x[k] + w * (x[k+1] - x[k]);
;     t += dt;
;   }
; }
;
; Register arguments:
;   s1 - n
;   s2 - t0
;   s3 - dt
;   s4 - x
;   s5 - y

lerp:
    bz    s1, 2f         ; Nothing to do?

    cpuid s6, z, z       ; Query maximum vector length
    mov   vl, s6

    add   s8, s4, #4     ; s8 = &x[1]
    itof  s7, s6, z

    ldea  v1, z, #1
    itof  v1, v1, z
    fmul  v1, v1, s3     ; v1 = dt * [0.0, 1.0, 2.0, ...]

    fmul  s7, s3, s7     ; s7 = dt * maximum vector length
1:
    minu  vl, s6, s1     ; Define the operation vector length
    sub   s1, s1, vl     ; Decrement loop counter

    ftoi  v2, v1, z      ; v2 = integer indexes (k)
    itof  v3, v2, z
    fsub  v3, v1, v3     ; v3 = interpolation weight (w)

    ldw   v4, s4, v2*4   ; Load x[k]
    ldw   v5, s8, v2*4   ; Load x[k+1]

    fsub  v5, v5, v4
    fmul  v5, v5, v3
    fadd  v5, v4, v5     ; v5 = x[k] + w * (x[k+1] - x[k])

    stw   v5, s5, #4     ; Store y (element stride = 4 bytes)

    ldea  s5, s5, vl*4   ; Increment address (y)
    fadd  v1, v1, s7     ; Increment t
    bnz   s1, 1b
2:
    ret
```

Bibliography

[1] ANSI/IEEE Std 754-2008, IEEE standard for floating-point arithmetic, 2008.