MRISC32 Instruction Set Manual

Version v0.4.0

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 simple yet powerful and highly scalable.

One of the main features of the instruction set architecture is its forward-looking vector functionality that not only integrates well with the rest of the ISA, but also enables implementations to freely select the level of hardware parallelism. This makes the vector functionality suitable for low-end systems like embedded microcontrollers as well as for high-performance computing (HPC).

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

1.2 Architecture modules

The MRISC32 instruction set architecture consists of the mandatory Base architecture, plus the following optional architecture modules:

- Vector operation module (VM)
- Packed operation module (PM)
- Floating-point module (FM)
- Saturating and halving arithmetic module (SM)

Each optional architecture module independently extends the capabilities of the instruction set.

1.3 Data types

1.3.1 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

1.3.2 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

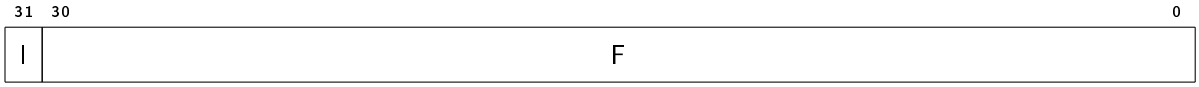
1.3.3 Fixed point types

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.

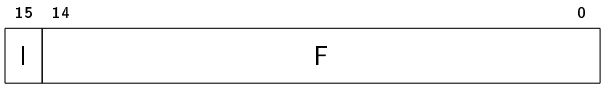
The value of a Q31 number is obtained by interpreting the bit vector as a two's complement signed integer multiplied by 2^{-31}.



Q15

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

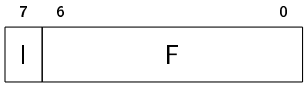
The value of a Q15 number is obtained by interpreting the bit vector as a two's complement signed integer multiplied by 2^{-15}.



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}.



1.3.4 Floating-point types

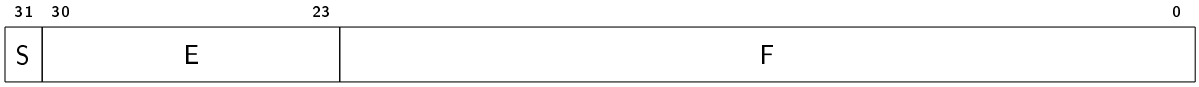
Name	Size	Meaning
float32	32	Single precision binary floating-point number
float16	16	Half precision binary floating-point number
float8	8	Quarter precision binary floating-point number

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.



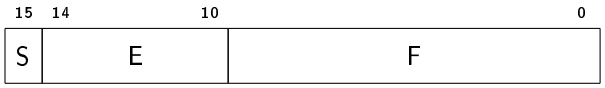
[1] The float32 type uses the same format and interpretation as IEEE 754-2008 binary32

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.

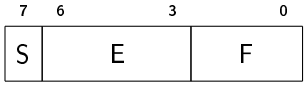


float8

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

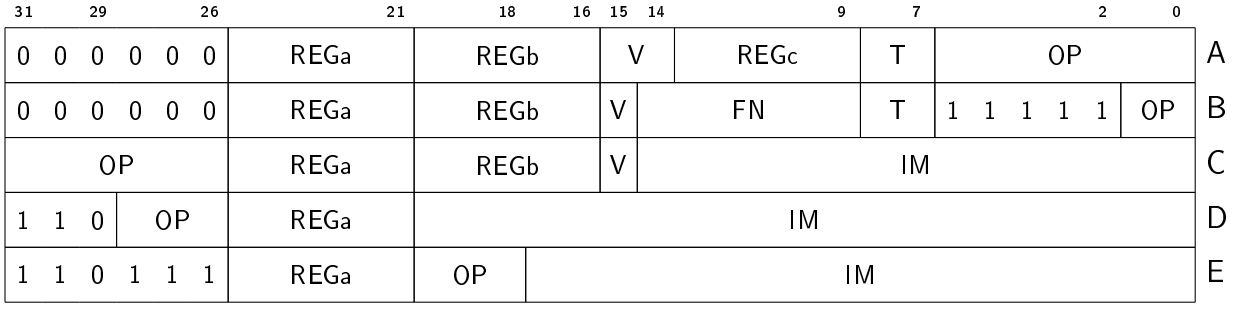
The significand has an implicit leading bit (to the left of the binary point) with value 1, giving four effective significand bits.

The exponent bias is 7.



1.4 Instruction encoding

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



For format A instructions, the value of the OP field must be in the range 0000100_2 to 1111011_2 (4_{10} to 123_{10}).

For format C instructions, the value of the OP field must be in the range 000100_2 to 101111_2 (4_{10} to 47_{10}).

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

For format D instructions, the value of the OP field must be in the range 000_2 to 110_2 (0_{10} to 6_{10}).

1.4.1 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
REGa	Destination/source register number (0-31)
REGb	Source register number (0-31)
REGc	Source register number (0-31)
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. Each OP, FN combination is referred to as a major instruction.

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_2	S,S[,S]	S,S,S
10_2	V,V[,S]	V,S,S
11_2	V,V,V	V,S,V
01_2	V,V,fold(V)	(reserved)

The register fields REGa, REGb and REGc 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 REG-field has the value 21_{10}, then the register operand is V21.

The first register operand, REGa, can be a source or a destination register depending on the instruction, while REGb and REGc 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 load/store instructions it defines a scaling factor for the register offset operand (i.e. the third operand):

T	Default	Load/store
00_2	One 32-bit word	*1
01_2	Four 8-bit bytes	*2
10_2	Two 16-bit half-words	*4
11_2	(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 field.

1.4.2 Format A

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

Format A can encode 120 major instructions (OP $\in$ $[4, 123]$).

1.4.3 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.

Format B can encode 256 major instructions (OP $\in$ $[0, 3]$, FN $\in$ $[0, 63]$).

1.4.4 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. For instance, the instruction ADD exists in both format A and format C encodings.

Format C can encode 44 major instructions (OP $\in$ $[4, 47]$).

1.4.5 Format D

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

Format D can encode 7 major instructions (OP $\in [0, 6]$).

1.4.6 Format E

Format E is used for conditional branch instructions.

Format E can encode 8 major instructions (OP $\in [0, 7]$).

1.4.7 Future extensions and encodings

The following table lists the actual and maximum number of instructions per instruction format:

Format	Count	Max	Used
A	78	120	65%
B	10	256	4%
C	37	44	84%
D	7	7	100%
E	8	8	100%

Encodings with the four most significant bits set to 1110 or 1111 are reserved for future encoding formats (or for extending the number of possible instructions for existing formats).

As can be seen, there is ample room for adding more instructions in future versions of the ISA.

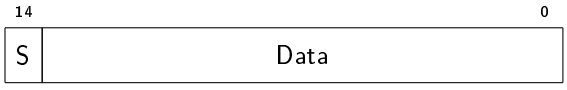
1.5 Immediate value encoding

The encoded width of an immediate operand depends on the instruction encoding format, and for each instruction format there is one or more possible interpretations (encodings) of the immediate operand (which encoding to use depends on the instruction).

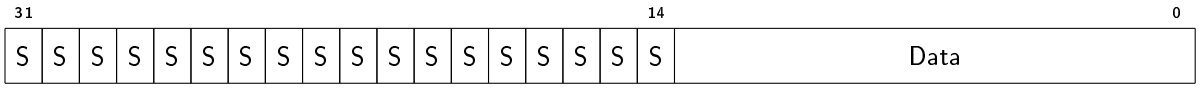
Format	Width	Immediate encodings
C	15 bits	I15, I15HL
D	21 bits	I21HL, I21H, I21X4
E	18 bits	I18X4

1.5.1 I15

The I15 format is encoded using 15 bits as follows:

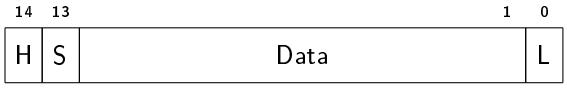


The immediate value is expanded into a 32-bit word as follows:

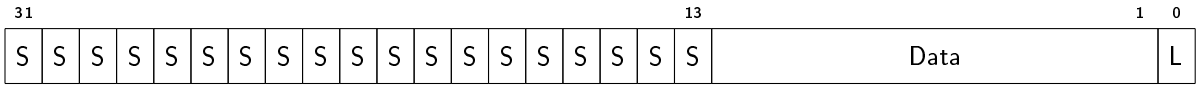


1.5.2 I15HL

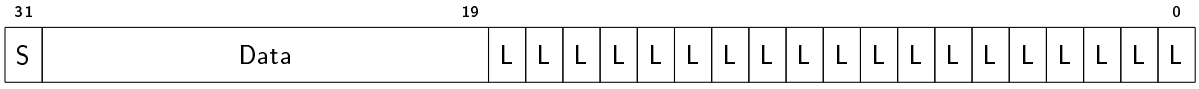
The I15HL format is encoded using 15 bits as follows:



When H=0, the immediate value is expanded into a 32-bit word as follows:

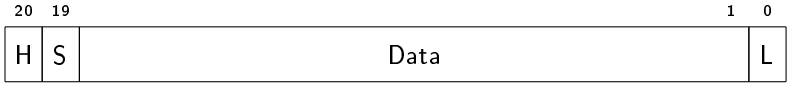


When H=1, the immediate value is expanded into a 32-bit word as follows:

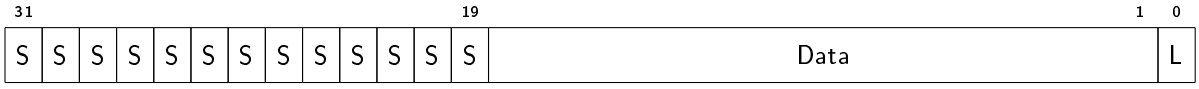


1.5.3 I21HL

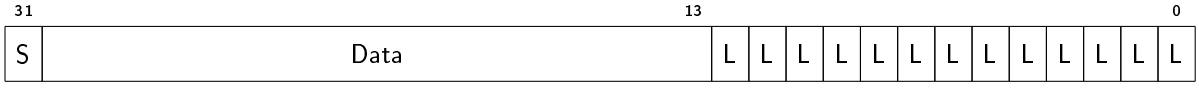
The I21HL format is encoded using 21 bits as follows:



When H=0, the immediate value is expanded into a 32-bit word as follows:

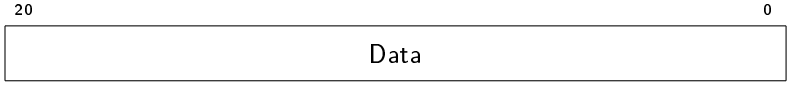


When H=1, the immediate value is expanded into a 32-bit word as follows:

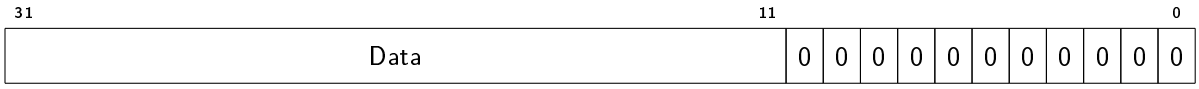


1.5.4 I21H

The I21H format is encoded using 21 bits as follows:

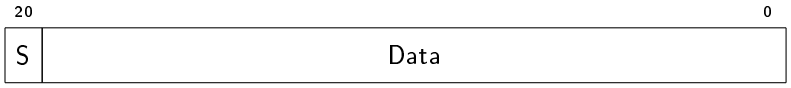


The immediate value is expanded into a 32-bit word as follows:

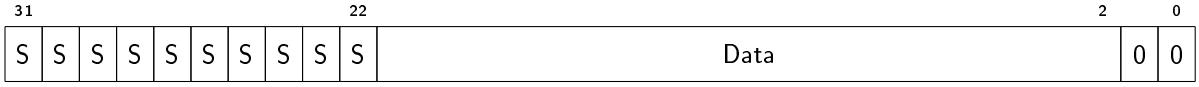


1.5.5 I21X4

The I21X4 format is encoded using 21 bits as follows:

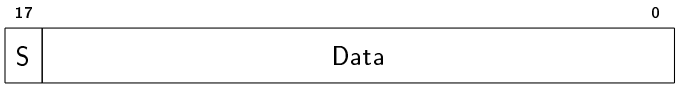


The immediate value is expanded into a 32-bit word as follows:

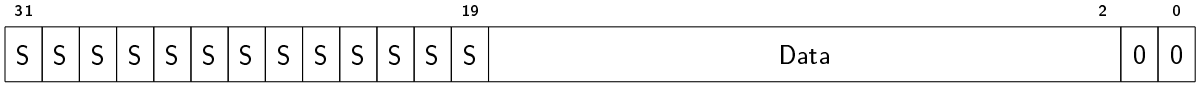


1.5.6 I18X4

The I18X4 format is encoded using 18 bits as follows:



The immediate value is expanded into a 32-bit word as follows:



1.6 Assembler syntax

TBD

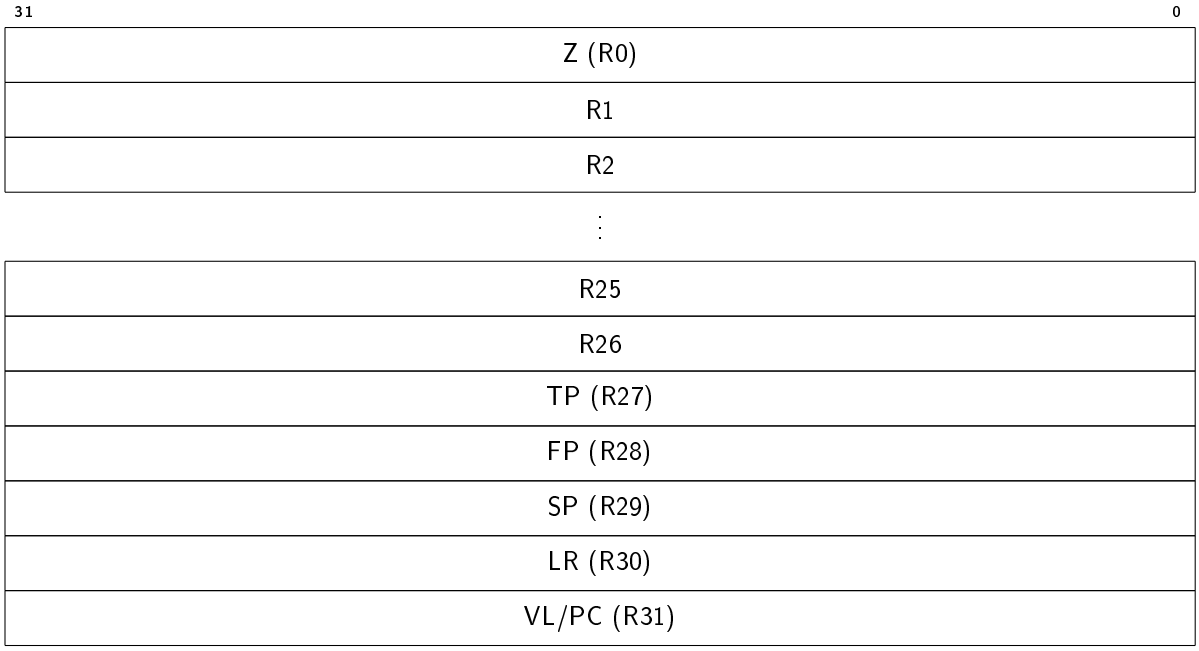
Chapter 2

Base architecture

The Base architecture is present in all implementations of the MRISC32 ISA. It primarily provides scalar integer and control flow instructions, and constitutes the minimum requirement for an MRISC32 implementation.

2.1 Scalar registers

There are 32 user addressable scalar registers, each 32 bits wide.



2.1.1 The Z register

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

2.1.2 The LR register

LR is the link register, which contains the return address for subroutines. It can also be used as a general purpose register.

2.1.3 The VL register

VL is the vector length register, which defines the length of vector operations. It can also be used as a general purpose register when its value is not used by any vector operations.

If an implementation does not supprot the Vector operation module (VM), the VL register acts as a regular general purpose register.

Please note that a select few instructions substitue the program counter for R31, which means that those instructions can not access the VL register.

2.1.4 TP, FP and SP

The scalar registers TP, FP and SP are aliases for R27, R28 and R29, respectively. They have no special architectural meaning, but it is recommended that they are used as follows:

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

The registers can also be used as general purpose registers.

For more information, see 10.

2.2 The program counter

The program counter (PC) is an internal register that holds the memory address of the current instruction.

The only instructions that can alter the PC register are control flow instructions (branches and jumps), that implicitly modify the program counter.

A few instructions substitue PC for R31 as a read-only operand, but most instructions can not address the PC register explicitly.

Furthermore, a few instructions use the value of the program counter as an implicit input operand.

2.3 Memory addressing

TBD

2.4 Exceptions

TBD

Chapter 3

Vector operation module (VM)

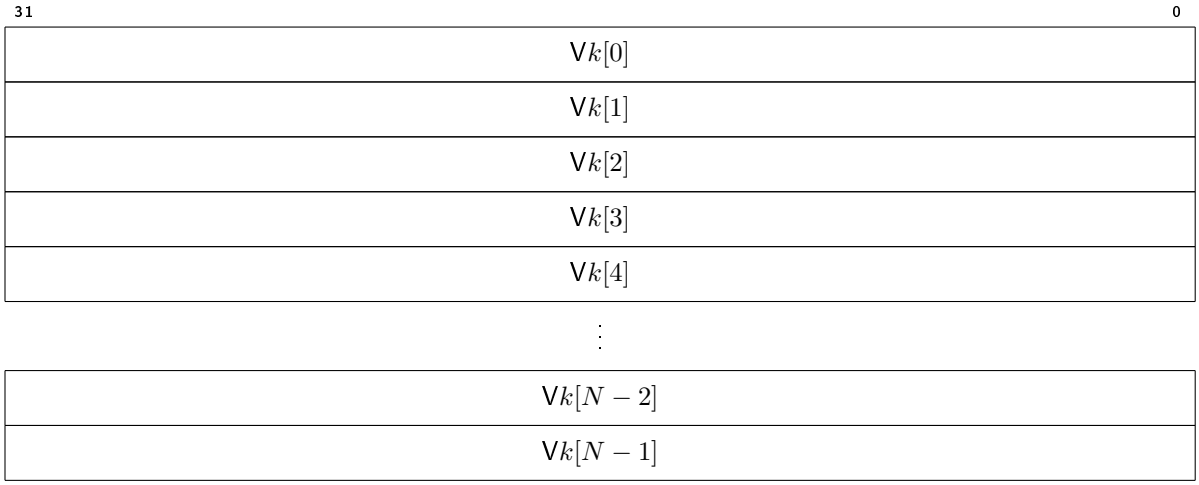
The Vector operation module adds facilities for vector processing. A set of vector registers is added, and most instructions are extended to support processing of vector registers.

3.1 Vector registers

There are 32 vector registers:

Vector reg. no	Name
0	VZ
1	V1
2	V2
3	V3
4	V4
⋮	
30	V30
31	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):



3.1.1 The VZ register

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

3.2 Vector operation

A vector operation is performed when a source or destination operand of an instruction is a vector register.

3.2.1 Vector length

The vector length is the number of vector elements to process in a vector operation.

All vector operations use the vector length that is given by the value of the VL register at the time of instruction invocation.

When the vector length is M, vector elements $[0, M)$ are processed.

To obtain the maximum vector length for the implementation, read the MAX_VL system register.

> **Note**
>
> The maximum vector length, as advertised by the MAX_VL system register, reflects the implementation dependent vector register size. By respecting the value of MAX_VL, software can be executed on implementations with different vector register sizes without modification.

> **TODO**
>
> *The vector length should be defined by the TBD vector register length (per vector register tag).*

3.2.2 Folding

Horizontal vector operations (e.g. sum and min/max) are supported by repeated folding, where the upper half of one vector source operand is combined with the lower half of another vector source operand.

> **TODO**
>
> *Describe how folding works.*

3.2.3 Masking

> **TODO**
>
> *Define and describe masked vector operations.*

3.2.4 Operation

A vector operation is performed as if all vector elements are processed as a series of scalar operations, in order from the lowest vector element index to the highest vector element index of the operation.

> **Note**
>
> An implementation may process several vector elements concurrently in order to increase the operation throughput, but it is not a requirement.

The following sections describe how a vector operation is executed for different operand configurations. In each description the following applies:

- `VL` is the vector length of the operation
- `operation` is the operation to perform, as described by the instruction
- `Va, Vb, Vc` are vector register operands
- `Rb, Rc` are scalar register operands
- `IMM` is a scalar immediate operand
- `scale` is the optional index scale operand for load/store (1, 2, 4 or 8)

Vector, Vector, Vector

```
for i in 0 to VL-1 do
  operation(Va[i], Vb[i], Vc[i])
```

Vector, Vector, Scalar register

```
for i in 0 to VL-1 do
  operation(Va[i], Vb[i], Rc)
```

Vector, Vector, Scalar immediate

```
for i in 0 to VL-1 do
  operation(Va[i], Vb[i], #IMM)
```

Vector, Scalar, Scalar register (load/store)

```
for i in 0 to VL-1 do
  operation(Va[i], Rb, i × Rc × scale)
```

Vector, Scalar, Scalar immediate (load/store)

```
for i in 0 to VL-1 do
  operation(Va[i], Rb, i × IMM)
```

Vector, Vector, Vector - Folding

```
for i in 0 to VL-1 do
  operation(Va[i], Vb[VL+i], Vc[i])
```

Chapter 4

Packed operation module (PM)

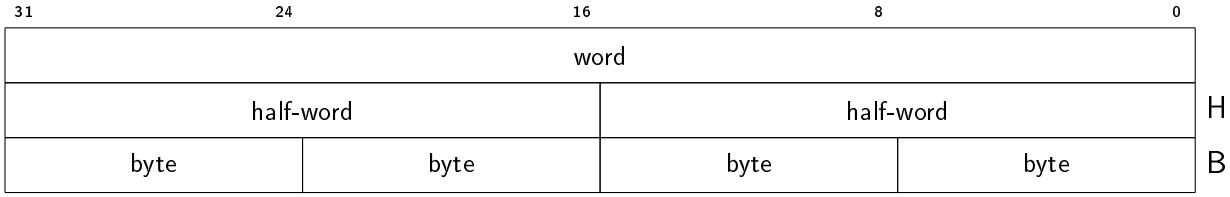
The Packed operation module adds facilities for parallel operation on packed data types. Most instructions are extended with packed operation modes, and a few instructions are added that mainly deal with packing and unpacking of data of different sizes.

Both scalar registers and vector registers may be used to hold packed data types.

4.1 Packed data operation

Many instructions are extended with the ability to operate on several individual sub-parts of the source and destination elements. These sub-parts are referred to as slices.

A single 32-bit element may be split up into one, two or four slices, as follows:



When a packed operation is performed, all slices within a 32-bit word are processed in parallel. It is not possible to process only a subset of the slices.

4.1.1 Word mode

In word mode, which is the default, each element is processed as a single 32-bit slice.

4.1.2 Half-word mode

In half-word mode each element is processed as two individual 16-bit slices in parallel.

In assembly language, half-word mode is indicated by appending the suffix .H to the instruction mnemonic.

4.1.3 Byte mode

In byte mode each element is processed as four individual 8-bit slices in parallel.

In assembly language, byte mode is indicated by appending the suffix .B to the instruction mnemonic.

4.1.4 Packed floating-point operation

For floating-point instructions, using packed operating modes implies using floating-point precisions lower than single precision:

Mode	Precision
word	Single precision floating-point
half-word	Half precision floating-point
byte	Quarter precision floating-point

Chapter 5

Floating-point module (FM)

The Floating-point module adds instructions that operate on floating-point numbers. Both scalar registers and vector registers may be used to hold floating-point values.

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

TBD

Chapter 6

Saturating and halving arithmetic module (SM)

The Saturating and halving arithmetic module adds instructions that extends the capabilities for operating on fixed point numbers.

TBD

Chapter 7

Instructions

This chapter describes all the instructions of the MRISC32 instruction set.

Instruction variants with a .B (packed byte) or .H (packed half-word) mnemonic suffix are only available in implementations that support the Packed operation module (PM).

Instruction variants that include vector register operands are only available in implementations that support the Vector operation module (VM).

For instructions that are not part of the Base architecture, the required architecture module (or modules) is indicated in the instruction documentation.

The encoding format used for immediate operands is documented per instruction (the IM field, if any, references the immediate encoding format).

Bits in the instruction encoding that are reserved are indicated in gray, and must be set to zero (0).

7.1 Pseudocode

The operation that an instruction performs is described using pseudocode.

7.1.1 Pseudocode scope

The pseudocode for each instruction shall be regarded as a function that is executed for *each slice* 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 slices and the size of each slice is dictated by the packed operation mode.

As an example, consider a byte mode instruction operating on a vector. In this case the pseudocode function is performed for each 8-bit slice of each 32-bit vector element, as shown in figure 7.1.

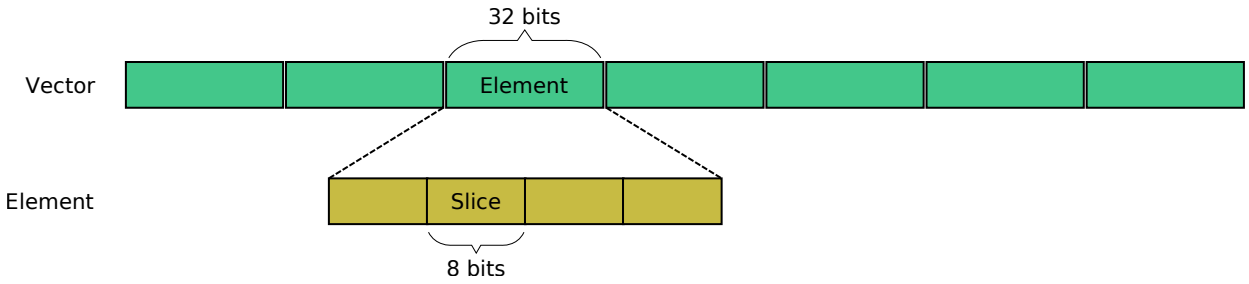


Figure 7.1: Example of an 8-bit slice within a vector element.

7.1.2 Types

bit vector

A vector of bits of a given size, without any particular interpretation of the meaning of the bits.

Instruction source and destination operands are treated as bit vectors. To perform arithmetic operations, a bit vector must first be interpreted as an integer or real value.

Example of an 8-bit bit vector: 00101101_2.

integer

An integer value in the range $(-\infty, +\infty)$.

Integers support integer arithmetic operations.

Example: -12345.

real

A real value in the range $(-\infty, +\infty)$, with infinite precision.

Real values support real arithmetic operations.

Example: -123.45.

7.1.3 Type conversions

Type conversions can either be explicit or implicit.

Explicit conversions are typically used for interpreting a bit vector as an integer or real value, e.g. in order to perform arithmetic opertaions. This can be done with pseudocode functions such as uint(x) and float(x).

Implicit conversions are used when interpreting an integer or real value as a bit vector, e.g. for assignment of the destination operand (which is always a bit vector) or when performing bitwise or shift operations on an integer value.

An implicit conversion to a bit vector is done as follows:

- Integer values are converted to a two's complement form bit vector of infinite width, which is then truncated to the target width.
- Real values are converted to an IEEE 754 binary bit vector representation of the target width.

7.1.4 Numeric constants

Unless otherwise noted, numeric constants are given as decimal (base 10) integers.

Integers in other bases are given as N_{base} (e.g. 101_2).

Real values are given in base 10 (e.g. 10.2).

7.1.5 Notation

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

Notation	Meaning
REGa, REGb, REGc	Register number fields of the instruction word
IM	IM field of the instruction word
T	T field of the instruction word
V	Vector mode (two bits)
a, b, c	1st, 2nd and 3rd operation operand (slice bit vectors)
bits	Slice size, in bits
scale	Scale factor according to the T field (1 for format C instructions)
i	Vector element number
x<k>	Bit k of bit vector x
x<k:l>	Bits k to l 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
SR[x]	System register number x
$\leftarrow$	Assignment
+, -	Addition, Subtraction
$\times$, /	Multiplication, Division
%	Remainder of integer division
$=$, $\neq$	Equal, Not equal
$<$, $>$	Less than, Greater than
$\leq$, $\geq$	Less than or equal, Greater than or equal
$\neg$, $\vee$, $\wedge$	Logical NOT, OR, AND
~, |, &, ^	Bitwise NOT, OR, AND, XOR
$<<$, $>>$	Zero-fill left-shift, right-shift
$<<_s$, $>>_s$	Sticky left-shift (fill with LSB), right-shift (fill with MSB)
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. Returns an integer.
uint(x)	Interpret bit vector x as an unsigned integer. Returns an integer.
float(x)	Interpret bit vector x as a floating-point number. Returns a real value.
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 bit vector x represents an IEEE 754 NaN value (not a number)
int2real(x)	Convert integer value to a real value
trunc(x)	Convert real value to an integer value, rounding towards zero (i.e. truncate)
round(x)	Convert real value to an integer value, rounding towards the nearest value (halfway cases are rounded away from zero)
pow(x, y)	Compute the value of x raised to the power y, i.e. x^y
crc32c(crc, b)	Starting with the initial value in crc, accumulate a CRC-32C value for the 8-bit integer in b (only the eight least significant bits of b are used).
crc32(crc, b)	Starting with the initial value in crc, accumulate a CRC-32 value for the 8-bit integer in b (only the eight least significant bits of b are used).

7.2 Load and store

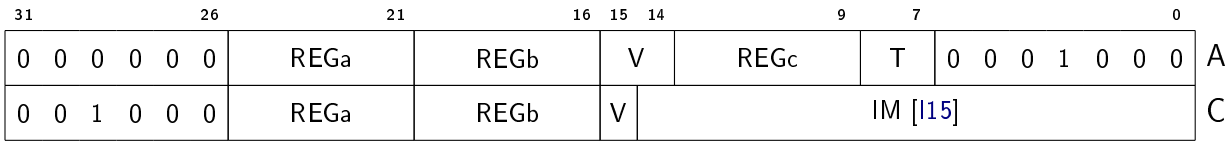
7.2.1 LDB - Load signed byte

Load and sign extend a byte (8 bits).

Operation

```
if V = 10₂ then
  adr ← int(b) + int(c) × i × scale
else
  adr ← int(b) + int(c) × scale
a ← int(MEM[adr,1])
```

Encoding



Variants

Fmt	V	T	Assembler
A	00_2	00_2	`LDB Ra, [Rb, Rc]`
A	00_2	01_2	`LDB Ra, [Rb, Rc*2]`
A	00_2	10_2	`LDB Ra, [Rb, Rc*4]`
A	00_2	11_2	`LDB Ra, [Rb, Rc*8]`
A	10_2	00_2	`LDB Va, [Rb, Rc]`
A	10_2	01_2	`LDB Va, [Rb, Rc*2]`
A	10_2	10_2	`LDB Va, [Rb, Rc*4]`
A	10_2	11_2	`LDB Va, [Rb, Rc*8]`
A	11_2	00_2	`LDB Va, [Rb, Vc]`
A	11_2	01_2	`LDB Va, [Rb, Vc*2]`
A	11_2	10_2	`LDB Va, [Rb, Vc*4]`
A	11_2	11_2	`LDB Va, [Rb, Vc*8]`
C	00_2	00_2	`LDB Ra, [Rb, #imm]`
C	10_2	00_2	`LDB Va, [Rb, #imm]`

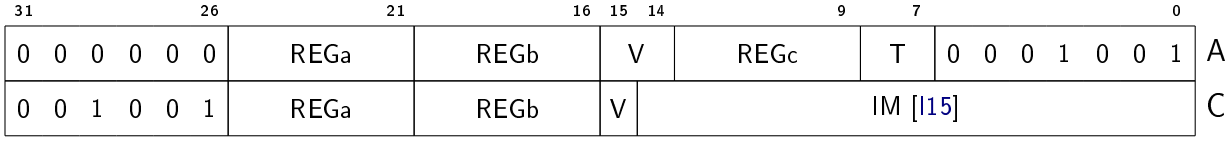
7.2.2 LDH - Load signed half-word

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

Operation

```
if V = 10₂ then
  adr ← int(b) + int(c) × i × scale
else
  adr ← int(b) + int(c) × scale
a ← int(MEM[adr,2])
```

Encoding



Variants

Fmt	V	T	Assembler
A	00_2	00_2	`LDH Ra, [Rb, Rc]`
A	00_2	01_2	`LDH Ra, [Rb, Rc*2]`
A	00_2	10_2	`LDH Ra, [Rb, Rc*4]`
A	00_2	11_2	`LDH Ra, [Rb, Rc*8]`
A	10_2	00_2	`LDH Va, [Rb, Rc]`
A	10_2	01_2	`LDH Va, [Rb, Rc*2]`
A	10_2	10_2	`LDH Va, [Rb, Rc*4]`
A	10_2	11_2	`LDH Va, [Rb, Rc*8]`
A	11_2	00_2	`LDH Va, [Rb, Vc]`
A	11_2	01_2	`LDH Va, [Rb, Vc*2]`
A	11_2	10_2	`LDH Va, [Rb, Vc*4]`
A	11_2	11_2	`LDH Va, [Rb, Vc*8]`
C	00_2	00_2	`LDH Ra, [Rb, #imm]`
C	10_2	00_2	`LDH Va, [Rb, #imm]`

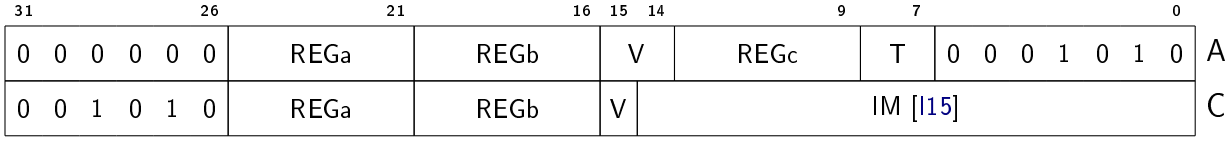
7.2.3 LDW - Load word

Load a word (32 bits).

Operation

```
if V = 10₂ then
  adr ← int(b) + int(c) × i × scale
else
  adr ← int(b) + int(c) × scale
a ← int(MEM[adr,4])
```

Encoding



Variants

Fmt	V	T	Assembler
A	00_2	00_2	`LDW Ra, [Rb, Rc]`
A	00_2	01_2	`LDW Ra, [Rb, Rc*2]`
A	00_2	10_2	`LDW Ra, [Rb, Rc*4]`
A	00_2	11_2	`LDW Ra, [Rb, Rc*8]`
A	10_2	00_2	`LDW Va, [Rb, Rc]`
A	10_2	01_2	`LDW Va, [Rb, Rc*2]`
A	10_2	10_2	`LDW Va, [Rb, Rc*4]`
A	10_2	11_2	`LDW Va, [Rb, Rc*8]`
A	11_2	00_2	`LDW Va, [Rb, Vc]`
A	11_2	01_2	`LDW Va, [Rb, Vc*2]`
A	11_2	10_2	`LDW Va, [Rb, Vc*4]`
A	11_2	11_2	`LDW Va, [Rb, Vc*8]`
C	00_2	00_2	`LDW Ra, [Rb, #imm]`
C	10_2	00_2	`LDW Va, [Rb, #imm]`

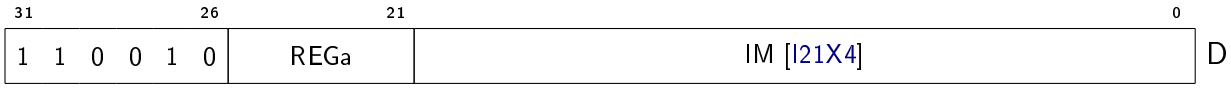
7.2.4 LDWPC - Load word PC-relative

Load a word (32 bits) from the address that is formed by adding the immediate value to the current PC.

Operation

```
adr ← int(PC) + int(b)
a ← int(MEM[adr,4])
```

Encoding



Variants

Assembler
`LDWPC Ra, #address@pc`

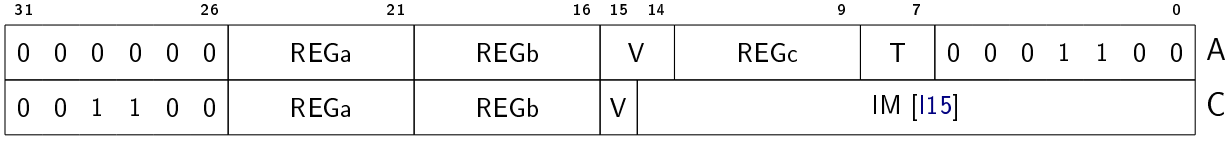
7.2.5 LDUB - Load unsigned byte

Load and zero extend a byte (8 bits).

Operation

```
if V = 10₂ then
  adr ← int(b) + int(c) × i × scale
else
  adr ← int(b) + int(c) × scale
a ← uint(MEM[adr,1])
```

Encoding



Variants

Fmt	V	T	Assembler
A	00_2	00_2	`LDUB Ra, [Rb, Rc]`
A	00_2	01_2	`LDUB Ra, [Rb, Rc*2]`
A	00_2	10_2	`LDUB Ra, [Rb, Rc*4]`
A	00_2	11_2	`LDUB Ra, [Rb, Rc*8]`
A	10_2	00_2	`LDUB Va, [Rb, Rc]`
A	10_2	01_2	`LDUB Va, [Rb, Rc*2]`
A	10_2	10_2	`LDUB Va, [Rb, Rc*4]`
A	10_2	11_2	`LDUB Va, [Rb, Rc*8]`
A	11_2	00_2	`LDUB Va, [Rb, Vc]`
A	11_2	01_2	`LDUB Va, [Rb, Vc*2]`
A	11_2	10_2	`LDUB Va, [Rb, Vc*4]`
A	11_2	11_2	`LDUB Va, [Rb, Vc*8]`
C	00_2	00_2	`LDUB Ra, [Rb, #imm]`
C	10_2	00_2	`LDUB Va, [Rb, #imm]`

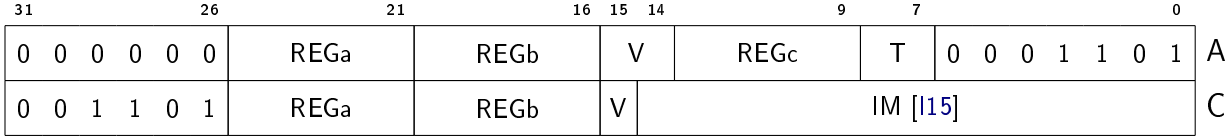
7.2.6 LDUH - Load unsigned half-word

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

Operation

```
if V = 10₂ then
  adr ← int(b) + int(c) × i × scale
else
  adr ← int(b) + int(c) × scale
a ← uint(MEM[adr,2])
```

Encoding



Variants

Fmt	V	T	Assembler
A	00_2	00_2	`LDUH Ra, [Rb, Rc]`
A	00_2	01_2	`LDUH Ra, [Rb, Rc*2]`
A	00_2	10_2	`LDUH Ra, [Rb, Rc*4]`
A	00_2	11_2	`LDUH Ra, [Rb, Rc*8]`
A	10_2	00_2	`LDUH Va, [Rb, Rc]`
A	10_2	01_2	`LDUH Va, [Rb, Rc*2]`
A	10_2	10_2	`LDUH Va, [Rb, Rc*4]`
A	10_2	11_2	`LDUH Va, [Rb, Rc*8]`
A	11_2	00_2	`LDUH Va, [Rb, Vc]`
A	11_2	01_2	`LDUH Va, [Rb, Vc*2]`
A	11_2	10_2	`LDUH Va, [Rb, Vc*4]`
A	11_2	11_2	`LDUH Va, [Rb, Vc*8]`
C	00_2	00_2	`LDUH Ra, [Rb, #imm]`
C	10_2	00_2	`LDUH Va, [Rb, #imm]`

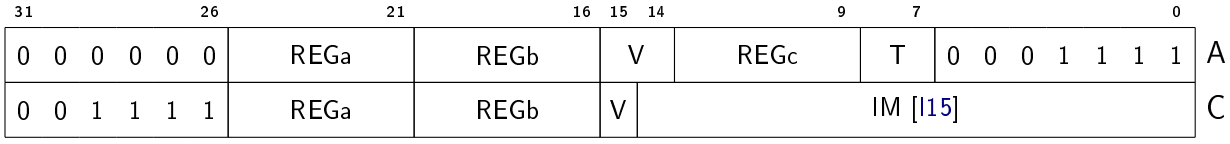
7.2.7 LDEA - Load effective address

Load effective address.

Operation

```
if V = 10₂ then
  a ← int(b) + int(c) × i × scale
else
  a ← int(b) + int(c) × scale
```

Encoding



Variants

Fmt	V	T	Assembler
A	00_2	00_2	`LDEA Ra, [Rb, Rc]`
A	00_2	01_2	`LDEA Ra, [Rb, Rc*2]`
A	00_2	10_2	`LDEA Ra, [Rb, Rc*4]`
A	00_2	11_2	`LDEA Ra, [Rb, Rc*8]`
A	10_2	00_2	`LDEA Va, [Rb, Rc]`
A	10_2	01_2	`LDEA Va, [Rb, Rc*2]`
A	10_2	10_2	`LDEA Va, [Rb, Rc*4]`
A	10_2	11_2	`LDEA Va, [Rb, Rc*8]`
A	11_2	00_2	`LDEA Va, [Rb, Vc]`
A	11_2	01_2	`LDEA Va, [Rb, Vc*2]`
A	11_2	10_2	`LDEA Va, [Rb, Vc*4]`
A	11_2	11_2	`LDEA Va, [Rb, Vc*8]`
C	00_2	00_2	`LDEA Ra, [Rb, #imm]`
C	10_2	00_2	`LDEA Va, [Rb, #imm]`

> **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.

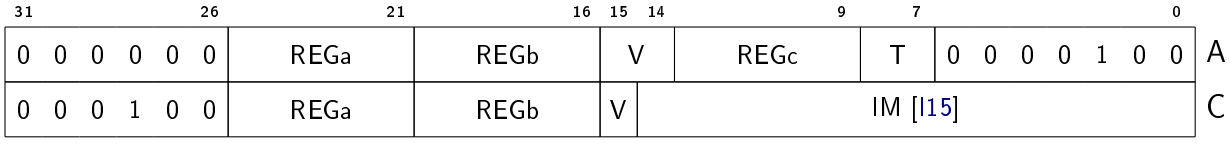
7.2.8 STB - Store byte

Store a byte (8 bits).

Operation

```
if V = 10₂ then
  adr ← int(b) + int(c) × i × scale
else
  adr ← int(b) + int(c) × scale
MEM[adr,1] ← a
```

Encoding



Variants

Fmt	V	T	Assembler
A	00_2	00_2	`STB Ra, [Rb, Rc]`
A	00_2	01_2	`STB Ra, [Rb, Rc*2]`
A	00_2	10_2	`STB Ra, [Rb, Rc*4]`
A	00_2	11_2	`STB Ra, [Rb, Rc*8]`
A	10_2	00_2	`STB Va, [Rb, Rc]`
A	10_2	01_2	`STB Va, [Rb, Rc*2]`
A	10_2	10_2	`STB Va, [Rb, Rc*4]`
A	10_2	11_2	`STB Va, [Rb, Rc*8]`
A	11_2	00_2	`STB Va, [Rb, Vc]`
A	11_2	01_2	`STB Va, [Rb, Vc*2]`
A	11_2	10_2	`STB Va, [Rb, Vc*4]`
A	11_2	11_2	`STB Va, [Rb, Vc*8]`
C	00_2	00_2	`STB Ra, [Rb, #imm]`
C	10_2	00_2	`STB Va, [Rb, #imm]`

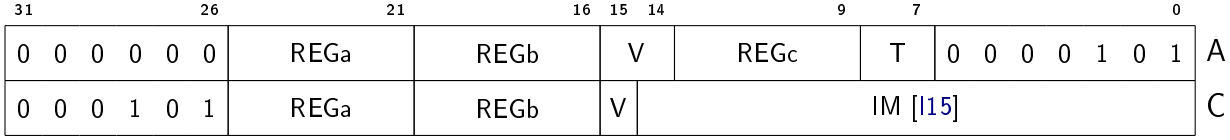
7.2.9 STH - Store half-word

Store a half-word (16 bits).

Operation

```
if V = 10₂ then
  adr ← int(b) + int(c) × i × scale
else
  adr ← int(b) + int(c) × scale
MEM[adr,2] ← a
```

Encoding



Variants

Fmt	V	T	Assembler
A	00_2	00_2	`STH Ra, [Rb, Rc]`
A	00_2	01_2	`STH Ra, [Rb, Rc*2]`
A	00_2	10_2	`STH Ra, [Rb, Rc*4]`
A	00_2	11_2	`STH Ra, [Rb, Rc*8]`
A	10_2	00_2	`STH Va, [Rb, Rc]`
A	10_2	01_2	`STH Va, [Rb, Rc*2]`
A	10_2	10_2	`STH Va, [Rb, Rc*4]`
A	10_2	11_2	`STH Va, [Rb, Rc*8]`
A	11_2	00_2	`STH Va, [Rb, Vc]`
A	11_2	01_2	`STH Va, [Rb, Vc*2]`
A	11_2	10_2	`STH Va, [Rb, Vc*4]`
A	11_2	11_2	`STH Va, [Rb, Vc*8]`
C	00_2	00_2	`STH Ra, [Rb, #imm]`
C	10_2	00_2	`STH Va, [Rb, #imm]`

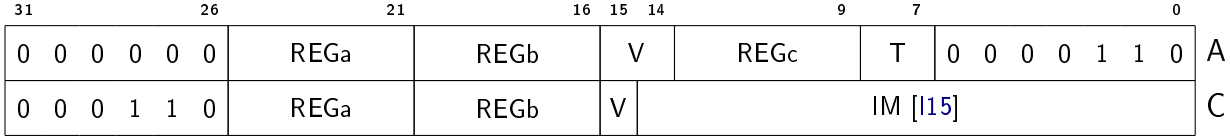
7.2.10 STW - Store word

Store a word (32 bits).

Operation

```
if V = 10₂ then
  adr ← int(b) + int(c) × i × scale
else
  adr ← int(b) + int(c) × scale
MEM[adr,4] ← a
```

Encoding



Variants

Fmt	V	T	Assembler
A	00_2	00_2	`STW Ra, [Rb, Rc]`
A	00_2	01_2	`STW Ra, [Rb, Rc*2]`
A	00_2	10_2	`STW Ra, [Rb, Rc*4]`
A	00_2	11_2	`STW Ra, [Rb, Rc*8]`
A	10_2	00_2	`STW Va, [Rb, Rc]`
A	10_2	01_2	`STW Va, [Rb, Rc*2]`
A	10_2	10_2	`STW Va, [Rb, Rc*4]`
A	10_2	11_2	`STW Va, [Rb, Rc*8]`
A	11_2	00_2	`STW Va, [Rb, Vc]`
A	11_2	01_2	`STW Va, [Rb, Vc*2]`
A	11_2	10_2	`STW Va, [Rb, Vc*4]`
A	11_2	11_2	`STW Va, [Rb, Vc*8]`
C	00_2	00_2	`STW Ra, [Rb, #imm]`
C	10_2	00_2	`STW Va, [Rb, #imm]`

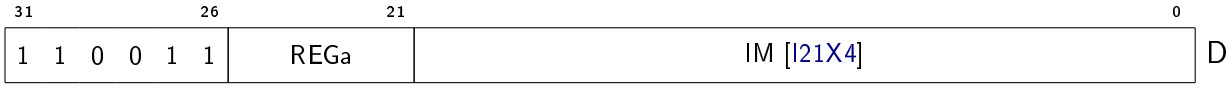
7.2.11 STWPC - Store word PC-relative

Store a word (32 bits) to the address that is formed by adding the immediate value to the current PC.

Operation

```
adr ← int(PC) + int(b)
MEM[adr,4] ← a
```

Encoding



Variants

Assembler
`STWPC Ra, #address@pc`

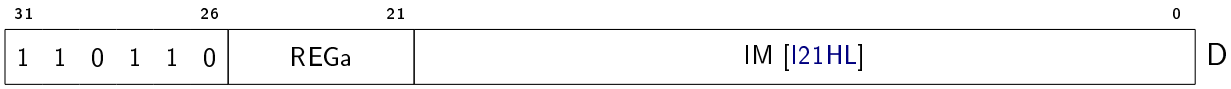
7.2.12 LDI - Load immediate

Load immediate value.

Operation

```
a ← b
```

Encoding



Variants

Assembler
`LDI Ra, #imm`

> **Note**
>
> With this instruction it is possible to load signed integer values in the range [-524288, 524287].
>
> It is also possible to load an immediate value that occupies the upper bits of a 32-bit word with the lower bits being filled with the LSB of the immediate field, making it suitable for loading values and masks such as 0x7fffffff and 0x8000ffff.
>
> This instruction can be used in combination with several instructions that take an immediate operand in order to form a full 32-bit value or absolute address. Examples of such instructions are OR, LDW and JL.
>
> Another use of this instruction is to load 32-bit floating-point values that can be represented with the 19 most significant bits (i.e. sign + exponent + 10 bits of mantissa), such as 1.0 (0x3f800000) or -255.0 (0xc37f0000).

7.3 Integer arithmetic

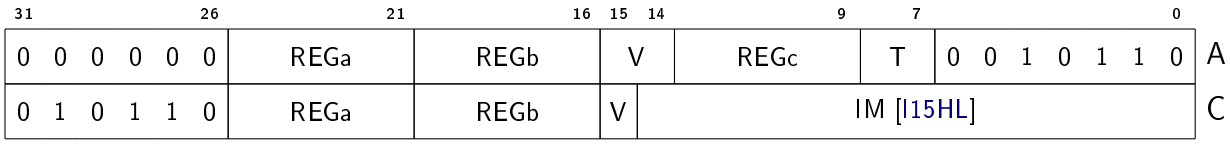
7.3.1 ADD - Add

Compute the sum of two integer operands.

Operation

```
a ← int(b) + int(c)
```

Encoding



Variants

Fmt	V	T	Assembler
A	00_2	00_2	`ADD Ra, Rb, Rc`
A	00_2	01_2	`ADD.B Ra, Rb, Rc`
A	00_2	10_2	`ADD.H Ra, Rb, Rc`
A	10_2	00_2	`ADD Va, Vb, Rc`
A	10_2	01_2	`ADD.B Va, Vb, Rc`
A	10_2	10_2	`ADD.H Va, Vb, Rc`
A	11_2	00_2	`ADD Va, Vb, Vc`
A	11_2	01_2	`ADD.B Va, Vb, Vc`
A	11_2	10_2	`ADD.H Va, Vb, Vc`
A	01_2	00_2	`ADD/F Va, Vb, Vc`
A	01_2	01_2	`ADD.B/F Va, Vb, Vc`
A	01_2	10_2	`ADD.H/F Va, Vb, Vc`
C	00_2	00_2	`ADD Ra, Rb, #imm`
C	10_2	00_2	`ADD Va, Vb, #imm`

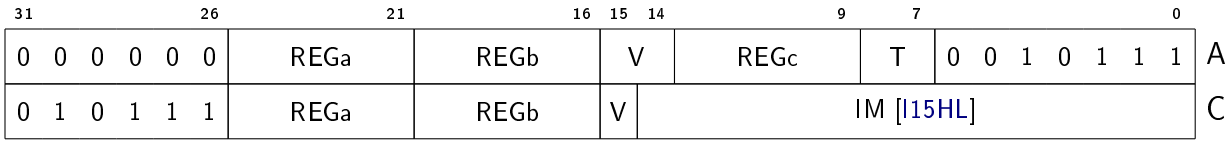
7.3.2 SUB - Subtract

Compute the difference of two integer operands.

Operation

```
a ← int(c) - int(b)
```

Encoding



Variants

Fmt	V	T	Assembler
A	00_2	00_2	`SUB Ra, Rc, Rb`
A	00_2	01_2	`SUB.B Ra, Rc, Rb`
A	00_2	10_2	`SUB.H Ra, Rc, Rb`
A	10_2	00_2	`SUB Va, Rc, Vb`
A	10_2	01_2	`SUB.B Va, Rc, Vb`
A	10_2	10_2	`SUB.H Va, Rc, Vb`
A	11_2	00_2	`SUB Va, Vc, Vb`
A	11_2	01_2	`SUB.B Va, Vc, Vb`
A	11_2	10_2	`SUB.H Va, Vc, Vb`
A	01_2	00_2	`SUB/F Va, Vc, Vb`
A	01_2	01_2	`SUB.B/F Va, Vc, Vb`
A	01_2	10_2	`SUB.H/F Va, Vc, Vb`
C	00_2	00_2	`SUB Ra, #imm, Rb`
C	10_2	00_2	`SUB Va, #imm, 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).

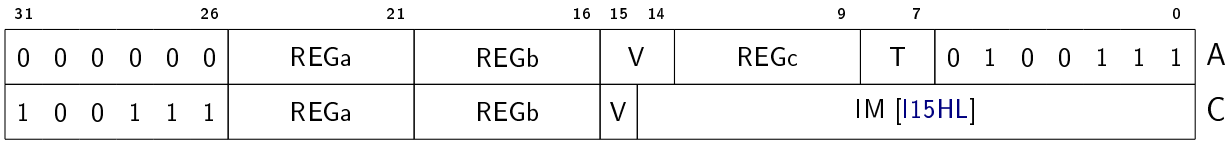
7.3.3 MUL - Multiply

Compute the product of two integer operands.

Operation

```
a ← int(b) × int(c)
```

Encoding



Variants

Fmt	V	T	Assembler
A	00_2	00_2	`MUL Ra, Rb, Rc`
A	00_2	01_2	`MUL.B Ra, Rb, Rc`
A	00_2	10_2	`MUL.H Ra, Rb, Rc`
A	10_2	00_2	`MUL Va, Vb, Rc`
A	10_2	01_2	`MUL.B Va, Vb, Rc`
A	10_2	10_2	`MUL.H Va, Vb, Rc`
A	11_2	00_2	`MUL Va, Vb, Vc`
A	11_2	01_2	`MUL.B Va, Vb, Vc`
A	11_2	10_2	`MUL.H Va, Vb, Vc`
A	01_2	00_2	`MUL/F Va, Vb, Vc`
A	01_2	01_2	`MUL.B/F Va, Vb, Vc`
A	01_2	10_2	`MUL.H/F Va, Vb, Vc`
C	00_2	00_2	`MUL Ra, Rb, #imm`
C	10_2	00_2	`MUL Va, Vb, #imm`

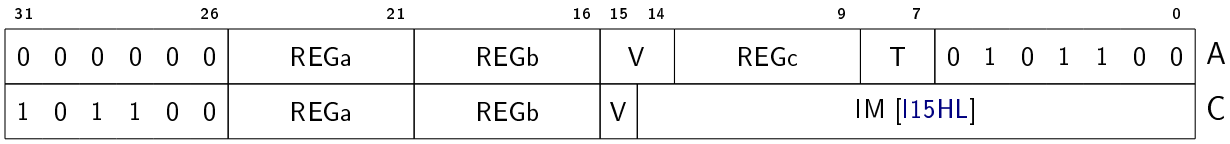
7.3.4 MADD - Multiply and add

Compute the product of two integer operands, and add the result to a third integer operand.

Operation

```
a ← int(a) + int(b) × int(c)
```

Encoding



Variants

Fmt	V	T	Assembler
A	00_2	00_2	`MADD Ra, Rb, Rc`
A	00_2	01_2	`MADD.B Ra, Rb, Rc`
A	00_2	10_2	`MADD.H Ra, Rb, Rc`
A	10_2	00_2	`MADD Va, Vb, Rc`
A	10_2	01_2	`MADD.B Va, Vb, Rc`
A	10_2	10_2	`MADD.H Va, Vb, Rc`
A	11_2	00_2	`MADD Va, Vb, Vc`
A	11_2	01_2	`MADD.B Va, Vb, Vc`
A	11_2	10_2	`MADD.H Va, Vb, Vc`
A	01_2	00_2	`MADD/F Va, Vb, Vc`
A	01_2	01_2	`MADD.B/F Va, Vb, Vc`
A	01_2	10_2	`MADD.H/F Va, Vb, Vc`
C	00_2	00_2	`MADD Ra, Rb, #imm`
C	10_2	00_2	`MADD Va, Vb, #imm`

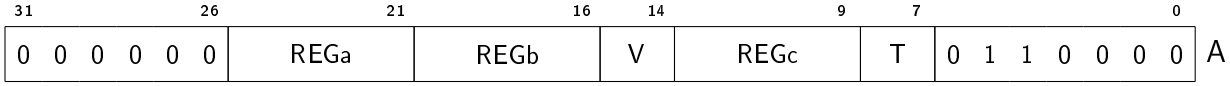
7.3.5 MULHI - Signed multiply high

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

Operation

```
a ← (int(b) × int(c)) >> bits
```

Encoding



Variants

V	T	Assembler
00_2	00_2	`MULHI Ra, Rb, Rc`
00_2	01_2	`MULHI.B Ra, Rb, Rc`
00_2	10_2	`MULHI.H Ra, Rb, Rc`
10_2	00_2	`MULHI Va, Vb, Rc`
10_2	01_2	`MULHI.B Va, Vb, Rc`
10_2	10_2	`MULHI.H Va, Vb, Rc`
11_2	00_2	`MULHI Va, Vb, Vc`
11_2	01_2	`MULHI.B Va, Vb, Vc`
11_2	10_2	`MULHI.H Va, Vb, Vc`
01_2	00_2	`MULHI/F Va, Vb, Vc`
01_2	01_2	`MULHI.B/F Va, Vb, Vc`
01_2	10_2	`MULHI.H/F Va, Vb, Vc`

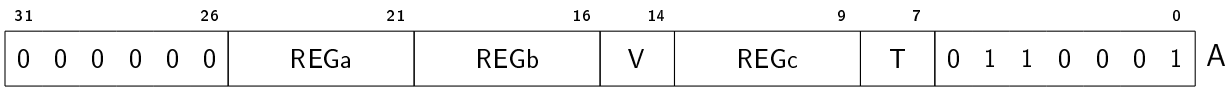
7.3.6 MULHIU - Unsigned multiply high

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

Operation

`a` $\leftarrow$ `(uint(b)` $\times$ `uint(c)) >> bits`

Encoding



Variants

V	T	Assembler
00_2	00_2	`MULHIU  Ra, Rb, Rc`
00_2	01_2	`MULHIU.B Ra, Rb, Rc`
00_2	10_2	`MULHIU.H Ra, Rb, Rc`
10_2	00_2	`MULHIU  Va, Vb, Rc`
10_2	01_2	`MULHIU.B Va, Vb, Rc`
10_2	10_2	`MULHIU.H Va, Vb, Rc`
11_2	00_2	`MULHIU  Va, Vb, Vc`
11_2	01_2	`MULHIU.B Va, Vb, Vc`
11_2	10_2	`MULHIU.H Va, Vb, Vc`
01_2	00_2	`MULHIU/F Va, Vb, Vc`
01_2	01_2	`MULHIU.B/F Va, Vb, Vc`
01_2	10_2	`MULHIU.H/F Va, Vb, Vc`

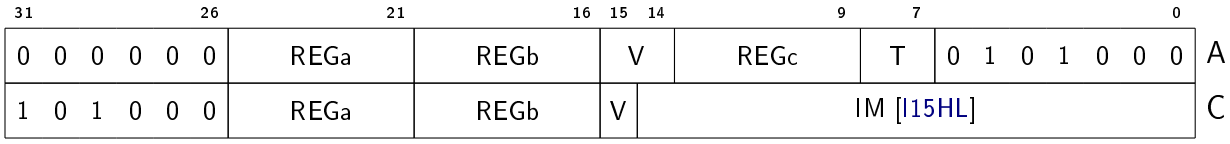
7.3.7 DIV - Signed divide

Compute the quotient of two signed integer operands.

Operation

`a ← int(b) / int(c)`

Encoding



Variants

Fmt	V	T	Assembler
A	00_2	00_2	`DIV Ra, Rb, Rc`
A	00_2	01_2	`DIV.B Ra, Rb, Rc`
A	00_2	10_2	`DIV.H Ra, Rb, Rc`
A	10_2	00_2	`DIV Va, Vb, Rc`
A	10_2	01_2	`DIV.B Va, Vb, Rc`
A	10_2	10_2	`DIV.H Va, Vb, Rc`
A	11_2	00_2	`DIV Va, Vb, Vc`
A	11_2	01_2	`DIV.B Va, Vb, Vc`
A	11_2	10_2	`DIV.H Va, Vb, Vc`
A	01_2	00_2	`DIV/F Va, Vb, Vc`
A	01_2	01_2	`DIV.B/F Va, Vb, Vc`
A	01_2	10_2	`DIV.H/F Va, Vb, Vc`
C	00_2	00_2	`DIV Ra, Rb, #imm`
C	10_2	00_2	`DIV Va, Vb, #imm`

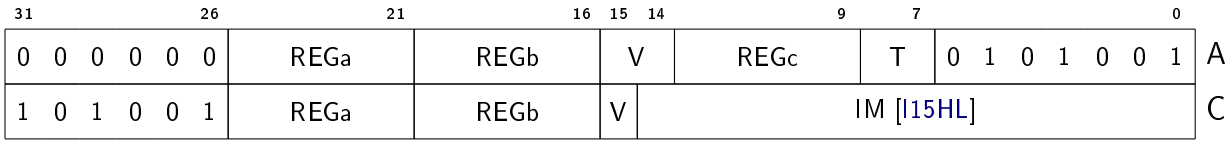
7.3.8 DIVU - Unsigned divide

Compute the quotient of two unsigned integer operands.

Operation

`a ← uint(b) / uint(c)`

Encoding



Variants

Fmt	V	T	Assembler
A	00_2	00_2	`DIVU Ra, Rb, Rc`
A	00_2	01_2	`DIVU.B Ra, Rb, Rc`
A	00_2	10_2	`DIVU.H Ra, Rb, Rc`
A	10_2	00_2	`DIVU Va, Vb, Rc`
A	10_2	01_2	`DIVU.B Va, Vb, Rc`
A	10_2	10_2	`DIVU.H Va, Vb, Rc`
A	11_2	00_2	`DIVU Va, Vb, Vc`
A	11_2	01_2	`DIVU.B Va, Vb, Vc`
A	11_2	10_2	`DIVU.H Va, Vb, Vc`
A	01_2	00_2	`DIVU/F Va, Vb, Vc`
A	01_2	01_2	`DIVU.B/F Va, Vb, Vc`
A	01_2	10_2	`DIVU.H/F Va, Vb, Vc`
C	00_2	00_2	`DIVU Ra, Rb, #imm`
C	10_2	00_2	`DIVU Va, Vb, #imm`

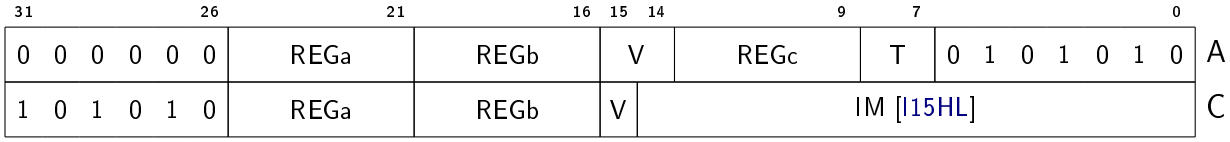
7.3.9 REM - Signed remainder

Compute the modulo of two signed integer operands.

Operation

`a ← int(b) % int(c)`

Encoding



Variants

Fmt	V	T	Assembler
A	00_2	00_2	`REM Ra, Rb, Rc`
A	00_2	01_2	`REM.B Ra, Rb, Rc`
A	00_2	10_2	`REM.H Ra, Rb, Rc`
A	10_2	00_2	`REM Va, Vb, Rc`
A	10_2	01_2	`REM.B Va, Vb, Rc`
A	10_2	10_2	`REM.H Va, Vb, Rc`
A	11_2	00_2	`REM Va, Vb, Vc`
A	11_2	01_2	`REM.B Va, Vb, Vc`
A	11_2	10_2	`REM.H Va, Vb, Vc`
A	01_2	00_2	`REM/F Va, Vb, Vc`
A	01_2	01_2	`REM.B/F Va, Vb, Vc`
A	01_2	10_2	`REM.H/F Va, Vb, Vc`
C	00_2	00_2	`REM Ra, Rb, #imm`
C	10_2	00_2	`REM Va, Vb, #imm`

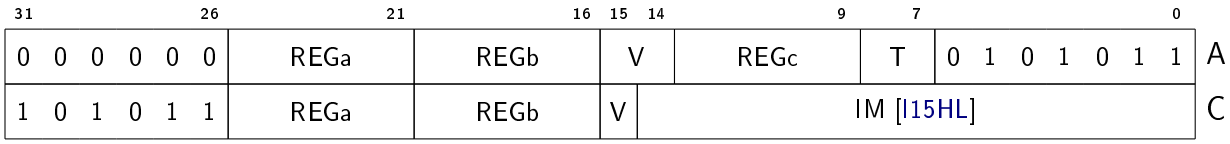
7.3.10 REMU - Unsigned remainder

Compute the modulo of two unsigned integer operands.

Operation

```
a ← uint(b) % uint(c)
```

Encoding



Variants

Fmt	V	T	Assembler
A	00_2	00_2	`REMU Ra, Rb, Rc`
A	00_2	01_2	`REMU.B Ra, Rb, Rc`
A	00_2	10_2	`REMU.H Ra, Rb, Rc`
A	10_2	00_2	`REMU Va, Vb, Rc`
A	10_2	01_2	`REMU.B Va, Vb, Rc`
A	10_2	10_2	`REMU.H Va, Vb, Rc`
A	11_2	00_2	`REMU Va, Vb, Vc`
A	11_2	01_2	`REMU.B Va, Vb, Vc`
A	11_2	10_2	`REMU.H Va, Vb, Vc`
A	01_2	00_2	`REMU/F Va, Vb, Vc`
A	01_2	01_2	`REMU.B/F Va, Vb, Vc`
A	01_2	10_2	`REMU.H/F Va, Vb, Vc`
C	00_2	00_2	`REMU Ra, Rb, #imm`
C	10_2	00_2	`REMU Va, Vb, #imm`

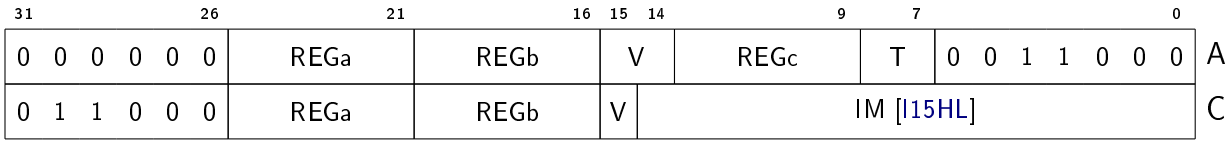
7.3.11 MIN - Signed minimum

Return the minimum value of two signed integer operands.

Operation

```
a ← min(int(b), int(c))
```

Encoding



Variants

Fmt	V	T	Assembler
A	00_2	00_2	`MIN Ra, Rb, Rc`
A	00_2	01_2	`MIN.B Ra, Rb, Rc`
A	00_2	10_2	`MIN.H Ra, Rb, Rc`
A	10_2	00_2	`MIN Va, Vb, Rc`
A	10_2	01_2	`MIN.B Va, Vb, Rc`
A	10_2	10_2	`MIN.H Va, Vb, Rc`
A	11_2	00_2	`MIN Va, Vb, Vc`
A	11_2	01_2	`MIN.B Va, Vb, Vc`
A	11_2	10_2	`MIN.H Va, Vb, Vc`
A	01_2	00_2	`MIN/F Va, Vb, Vc`
A	01_2	01_2	`MIN.B/F Va, Vb, Vc`
A	01_2	10_2	`MIN.H/F Va, Vb, Vc`
C	00_2	00_2	`MIN Ra, Rb, #imm`
C	10_2	00_2	`MIN Va, Vb, #imm`

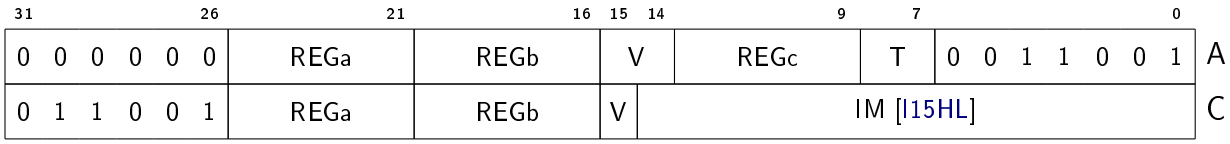
7.3.12 MAX - Signed maxiumum

Return the maximum value of two signed integer operands.

Operation

`a ← max(int(b), int(c))`

Encoding



Variants

Fmt	V	T	Assembler
A	00_2	00_2	`MAX Ra, Rb, Rc`
A	00_2	01_2	`MAX.B Ra, Rb, Rc`
A	00_2	10_2	`MAX.H Ra, Rb, Rc`
A	10_2	00_2	`MAX Va, Vb, Rc`
A	10_2	01_2	`MAX.B Va, Vb, Rc`
A	10_2	10_2	`MAX.H Va, Vb, Rc`
A	11_2	00_2	`MAX Va, Vb, Vc`
A	11_2	01_2	`MAX.B Va, Vb, Vc`
A	11_2	10_2	`MAX.H Va, Vb, Vc`
A	01_2	00_2	`MAX/F Va, Vb, Vc`
A	01_2	01_2	`MAX.B/F Va, Vb, Vc`
A	01_2	10_2	`MAX.H/F Va, Vb, Vc`
C	00_2	00_2	`MAX Ra, Rb, #imm`
C	10_2	00_2	`MAX Va, Vb, #imm`

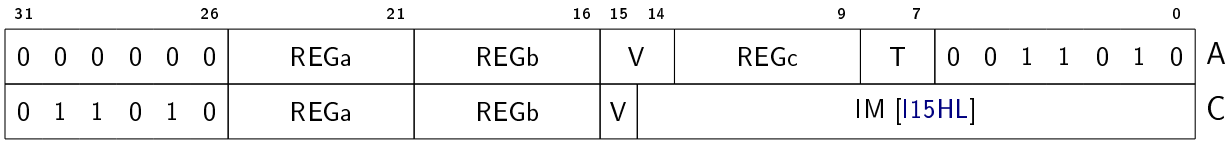
7.3.13 MINU - Unsigned minimum

Return the minimum value of two unsigned integer operands.

Operation

```
a ← min(uint(b), uint(c))
```

Encoding



Variants

Fmt	V	T	Assembler
A	00_2	00_2	`MINU Ra, Rb, Rc`
A	00_2	01_2	`MINU.B Ra, Rb, Rc`
A	00_2	10_2	`MINU.H Ra, Rb, Rc`
A	10_2	00_2	`MINU Va, Vb, Rc`
A	10_2	01_2	`MINU.B Va, Vb, Rc`
A	10_2	10_2	`MINU.H Va, Vb, Rc`
A	11_2	00_2	`MINU Va, Vb, Vc`
A	11_2	01_2	`MINU.B Va, Vb, Vc`
A	11_2	10_2	`MINU.H Va, Vb, Vc`
A	01_2	00_2	`MINU/F Va, Vb, Vc`
A	01_2	01_2	`MINU.B/F Va, Vb, Vc`
A	01_2	10_2	`MINU.H/F Va, Vb, Vc`
C	00_2	00_2	`MINU Ra, Rb, #imm`
C	10_2	00_2	`MINU Va, Vb, #imm`

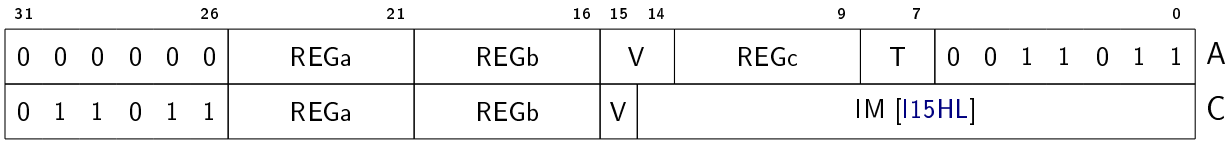
7.3.14 MAXU - Unsigned maximum

Return the maximum value of two unsigned integer operands.

Operation

```
a ← max(uint(b), uint(c))
```

Encoding



Variants

Fmt	V	T	Assembler
A	00_2	00_2	`MAXU Ra, Rb, Rc`
A	00_2	01_2	`MAXU.B Ra, Rb, Rc`
A	00_2	10_2	`MAXU.H Ra, Rb, Rc`
A	10_2	00_2	`MAXU Va, Vb, Rc`
A	10_2	01_2	`MAXU.B Va, Vb, Rc`
A	10_2	10_2	`MAXU.H Va, Vb, Rc`
A	11_2	00_2	`MAXU Va, Vb, Vc`
A	11_2	01_2	`MAXU.B Va, Vb, Vc`
A	11_2	10_2	`MAXU.H Va, Vb, Vc`
A	01_2	00_2	`MAXU/F Va, Vb, Vc`
A	01_2	01_2	`MAXU.B/F Va, Vb, Vc`
A	01_2	10_2	`MAXU.H/F Va, Vb, Vc`
C	00_2	00_2	`MAXU Ra, Rb, #imm`
C	10_2	00_2	`MAXU Va, Vb, #imm`

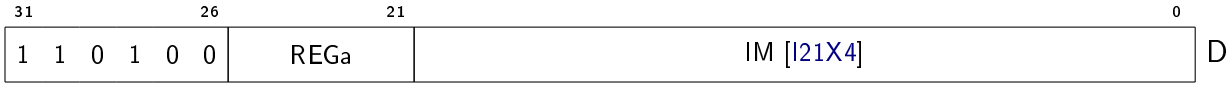
7.3.15 ADDPC - Add PC and immediate

Compute the sum of the current PC and an immediate operand.

Operation

```
a ← int(PC) + int(b)
```

Encoding



Variants

Assembler
`ADDPC Ra, #target@pc`

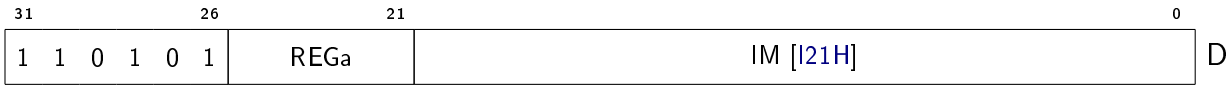
7.3.16 ADDPCHI - Add PC and high immediate

Compute the sum of the current PC and an immediate operand.

Operation

```
a ← int(PC) + int(b)
```

Encoding



Variants

Assembler
`ADDPCHI Ra, #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, LDH and JL.

7.4 Integer comparison

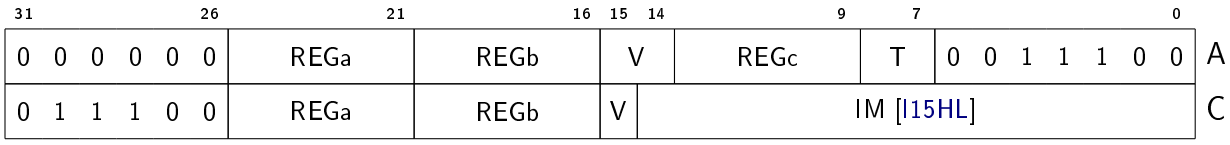
7.4.1 SEQ - Set if equal

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.

Operation

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

Encoding



Variants

Fmt	V	T	Assembler
A	00_2	00_2	`SEQ Ra, Rb, Rc`
A	00_2	01_2	`SEQ.B Ra, Rb, Rc`
A	00_2	10_2	`SEQ.H Ra, Rb, Rc`
A	10_2	00_2	`SEQ Va, Vb, Rc`
A	10_2	01_2	`SEQ.B Va, Vb, Rc`
A	10_2	10_2	`SEQ.H Va, Vb, Rc`
A	11_2	00_2	`SEQ Va, Vb, Vc`
A	11_2	01_2	`SEQ.B Va, Vb, Vc`
A	11_2	10_2	`SEQ.H Va, Vb, Vc`
A	01_2	00_2	`SEQ/F Va, Vb, Vc`
A	01_2	01_2	`SEQ.B/F Va, Vb, Vc`
A	01_2	10_2	`SEQ.H/F Va, Vb, Vc`
C	00_2	00_2	`SEQ Ra, Rb, #imm`
C	10_2	00_2	`SEQ Va, Vb, #imm`

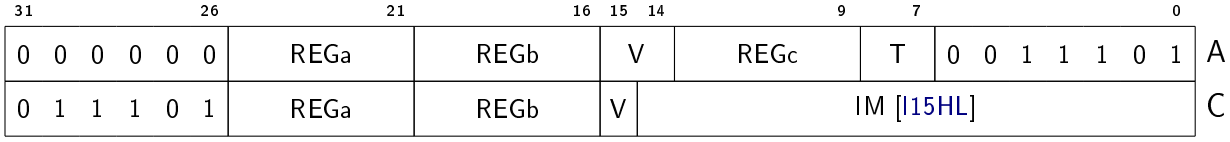
7.4.2 SNE - Set if not equal

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.

Operation

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

Encoding



Variants

Fmt	V	T	Assembler
A	00_2	00_2	`SNE Ra, Rb, Rc`
A	00_2	01_2	`SNE.B Ra, Rb, Rc`
A	00_2	10_2	`SNE.H Ra, Rb, Rc`
A	10_2	00_2	`SNE Va, Vb, Rc`
A	10_2	01_2	`SNE.B Va, Vb, Rc`
A	10_2	10_2	`SNE.H Va, Vb, Rc`
A	11_2	00_2	`SNE Va, Vb, Vc`
A	11_2	01_2	`SNE.B Va, Vb, Vc`
A	11_2	10_2	`SNE.H Va, Vb, Vc`
A	01_2	00_2	`SNE/F Va, Vb, Vc`
A	01_2	01_2	`SNE.B/F Va, Vb, Vc`
A	01_2	10_2	`SNE.H/F Va, Vb, Vc`
C	00_2	00_2	`SNE Ra, Rb, #imm`
C	10_2	00_2	`SNE Va, Vb, #imm`

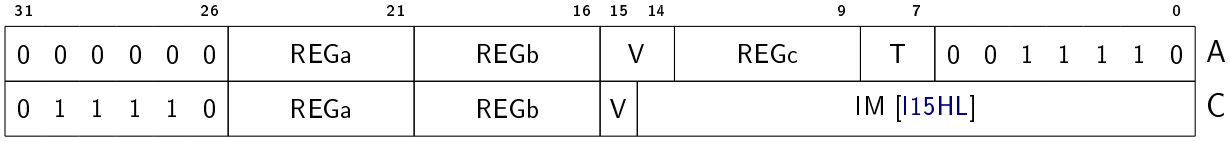
7.4.3 SLT - Set if less than

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.

Operation

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

Encoding



Variants

Fmt	V	T	Assembler
A	00_2	00_2	`SLT Ra, Rb, Rc`
A	00_2	01_2	`SLT.B Ra, Rb, Rc`
A	00_2	10_2	`SLT.H Ra, Rb, Rc`
A	10_2	00_2	`SLT Va, Vb, Rc`
A	10_2	01_2	`SLT.B Va, Vb, Rc`
A	10_2	10_2	`SLT.H Va, Vb, Rc`
A	11_2	00_2	`SLT Va, Vb, Vc`
A	11_2	01_2	`SLT.B Va, Vb, Vc`
A	11_2	10_2	`SLT.H Va, Vb, Vc`
A	01_2	00_2	`SLT/F Va, Vb, Vc`
A	01_2	01_2	`SLT.B/F Va, Vb, Vc`
A	01_2	10_2	`SLT.H/F Va, Vb, Vc`
C	00_2	00_2	`SLT Ra, Rb, #imm`
C	10_2	00_2	`SLT Va, Vb, #imm`

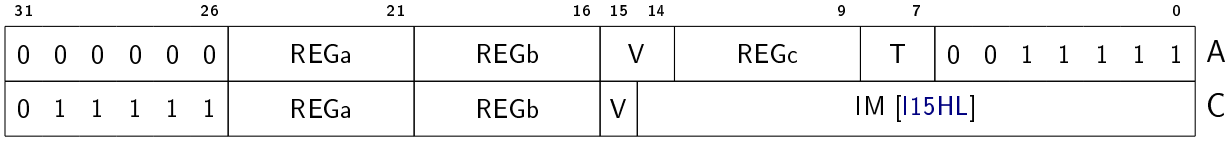
7.4.4 SLTU - Set if less than unsigned

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.

Operation

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

Encoding



Variants

Fmt	V	T	Assembler
A	00_2	00_2	`SLTU Ra, Rb, Rc`
A	00_2	01_2	`SLTU.B Ra, Rb, Rc`
A	00_2	10_2	`SLTU.H Ra, Rb, Rc`
A	10_2	00_2	`SLTU Va, Vb, Rc`
A	10_2	01_2	`SLTU.B Va, Vb, Rc`
A	10_2	10_2	`SLTU.H Va, Vb, Rc`
A	11_2	00_2	`SLTU Va, Vb, Vc`
A	11_2	01_2	`SLTU.B Va, Vb, Vc`
A	11_2	10_2	`SLTU.H Va, Vb, Vc`
A	01_2	00_2	`SLTU/F Va, Vb, Vc`
A	01_2	01_2	`SLTU.B/F Va, Vb, Vc`
A	01_2	10_2	`SLTU.H/F Va, Vb, Vc`
C	00_2	00_2	`SLTU Ra, Rb, #imm`
C	10_2	00_2	`SLTU Va, Vb, #imm`

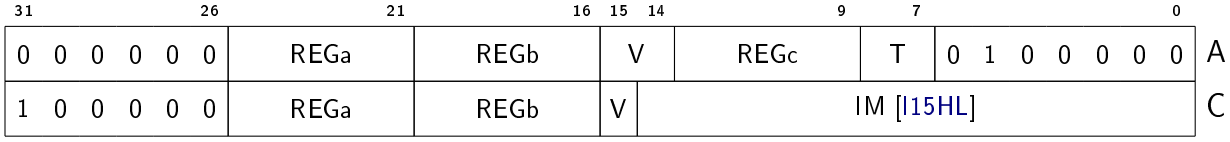
7.4.5 SLE - Set if less than or equal

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.

Operation

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

Encoding



Variants

Fmt	V	T	Assembler
A	00_2	00_2	`SLE Ra, Rb, Rc`
A	00_2	01_2	`SLE.B Ra, Rb, Rc`
A	00_2	10_2	`SLE.H Ra, Rb, Rc`
A	10_2	00_2	`SLE Va, Vb, Rc`
A	10_2	01_2	`SLE.B Va, Vb, Rc`
A	10_2	10_2	`SLE.H Va, Vb, Rc`
A	11_2	00_2	`SLE Va, Vb, Vc`
A	11_2	01_2	`SLE.B Va, Vb, Vc`
A	11_2	10_2	`SLE.H Va, Vb, Vc`
A	01_2	00_2	`SLE/F Va, Vb, Vc`
A	01_2	01_2	`SLE.B/F Va, Vb, Vc`
A	01_2	10_2	`SLE.H/F Va, Vb, Vc`
C	00_2	00_2	`SLE Ra, Rb, #imm`
C	10_2	00_2	`SLE Va, Vb, #imm`

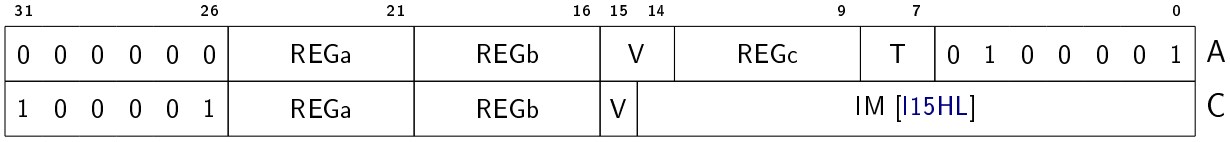
7.4.6 SLEU - Set if less than or equal unsigned

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.

Operation

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

Encoding



Variants

Fmt	V	T	Assembler
A	00_2	00_2	`SLEU Ra, Rb, Rc`
A	00_2	01_2	`SLEU.B Ra, Rb, Rc`
A	00_2	10_2	`SLEU.H Ra, Rb, Rc`
A	10_2	00_2	`SLEU Va, Vb, Rc`
A	10_2	01_2	`SLEU.B Va, Vb, Rc`
A	10_2	10_2	`SLEU.H Va, Vb, Rc`
A	11_2	00_2	`SLEU Va, Vb, Vc`
A	11_2	01_2	`SLEU.B Va, Vb, Vc`
A	11_2	10_2	`SLEU.H Va, Vb, Vc`
A	01_2	00_2	`SLEU/F Va, Vb, Vc`
A	01_2	01_2	`SLEU.B/F Va, Vb, Vc`
A	01_2	10_2	`SLEU.H/F Va, Vb, Vc`
C	00_2	00_2	`SLEU Ra, Rb, #imm`
C	10_2	00_2	`SLEU Va, Vb, #imm`

7.5 Branch

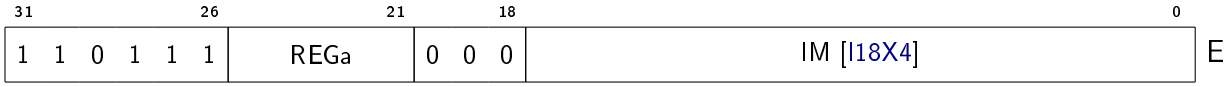
7.5.1 BZ - Branch if zero

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

Operation

```
if a = zeros(32) then
  PC ← int(PC) + int(b)
```

Encoding



Variants

Assembler
BZ Ra, #target

7.5.2 BNZ - Branch if not zero

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

Operation

```
if a ≠ zeros(32) then
  PC ← int(PC) + int(b)
```

Encoding

Variants

Assembler
`BNZ Ra, #target`

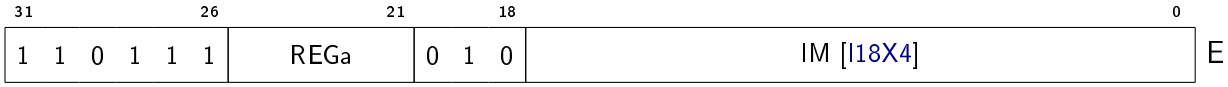
7.5.3 BS - Branch if set

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

Operation

```
if a = ones(32) then
  PC ← int(PC) + int(b)
```

Encoding



Variants

Assembler
`BS Ra, #target`

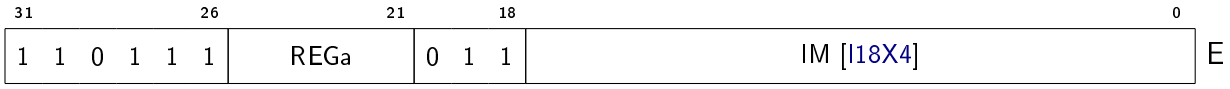
7.5.4 BNS - Branch if not set

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

Operation

```
if a ≠ ones(32) then
  PC ← int(PC) + int(b)
```

Encoding



Variants

Assembler
`BNS Ra, #target`

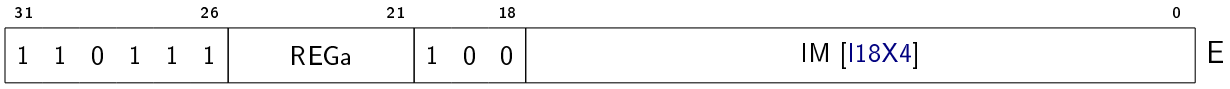
7.5.5 BLT - Branch if less than

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

Operation

```
if int(a) < 0 then
  PC ← int(PC) + int(b)
```

Encoding



Variants

Assembler
`BLT Ra, #target`

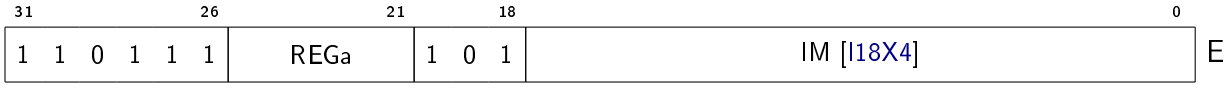
7.5.6 BGE - Branch if greater than or equal

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

Operation

```
if int(a) ≥ 0 then
  PC ← int(PC) + int(b)
```

Encoding



Variants

Assembler
BGE Ra, #target

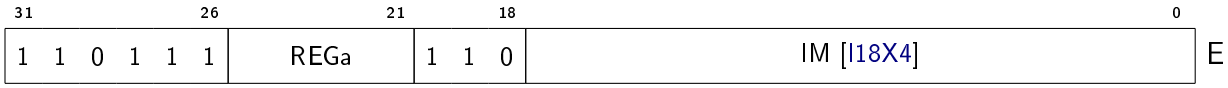
7.5.7 BLE - Branch if less than or equal

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

Operation

```
if int(a) ≤ 0 then
  PC ← int(PC) + int(b)
```

Encoding



Variants

Assembler
BLE Ra, #target

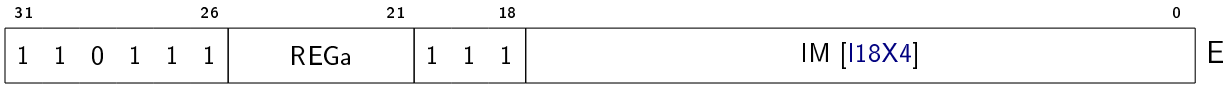
7.5.8 BGT - Branch if greater than

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

Operation

```
if int(a) > 0 then
  PC ← int(PC) + int(b)
```

Encoding



Variants

Assembler	
BGT	Ra, #target

7.5.9 J - Jump

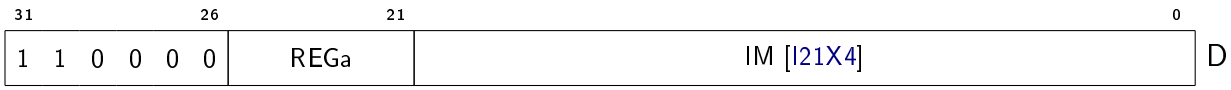
Jump to the target address that is formed by computing the sum of the register operand and the immediate operand.

As a special case, the register operand can be the program counter (PC), which is encoded as register number 31. This also means that the VL register can not be used as the register operand.

Operation

```
if REGa = 11111₂ then
  base ← PC
else
  base ← a
PC ← int(base) + int(b)
```

Encoding



Variants

Assembler
`J Ra, #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 LDI.
>
> 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).

7.5.10 JL - Jump and link

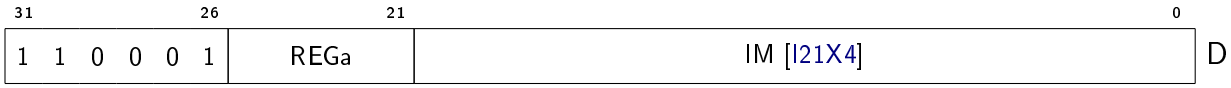
Jump and link. The current value of PC plus four is stored in the LR register, and the new PC is set to the target address that is formed by computing the sum of the register operand and immediate operand.

As a special case, the register operand can be the program counter (PC), which is encoded as register number 31. This also means that the VL register can not be used as the register operand.

Operation

```
if REGa = 11111₂ then
  base ← PC
else
  base ← a
LR ← int(PC) + 4
PC ← int(base) + int(b)
```

Encoding



Variants

Assembler
`JL Ra, #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 LDI.

7.6 Bitwise logic

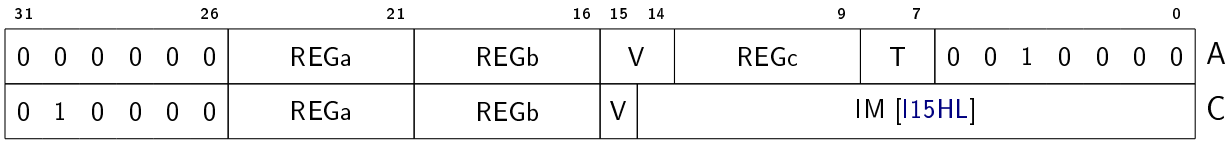
7.6.1 AND - Bitwise and

Compute the bitwise and of two integer operands, with optional negation of the source operands.

Operation

```
if T = 00₂ then
  a ← b & c
else if T = 01₂ then  // .PN
  a ← b & ~c
else if T = 10₂ then  // .NP
  a ← ~b & c
else if T = 11₂ then  // .NN
  a ← ~b & ~c
```

Encoding



Variants

Fmt	V	T	Assembler
A	00_2	00_2	`AND Ra, Rb, Rc`
A	00_2	01_2	`AND.PN Ra, Rb, Rc`
A	00_2	10_2	`AND.NP Ra, Rb, Rc`
A	00_2	11_2	`AND.NN Ra, Rb, Rc`
A	10_2	00_2	`AND Va, Vb, Rc`
A	10_2	01_2	`AND.PN Va, Vb, Rc`
A	10_2	10_2	`AND.NP Va, Vb, Rc`
A	10_2	11_2	`AND.NN Va, Vb, Rc`
A	11_2	00_2	`AND Va, Vb, Vc`
A	11_2	01_2	`AND.PN Va, Vb, Vc`
A	11_2	10_2	`AND.NP Va, Vb, Vc`
A	11_2	11_2	`AND.NN Va, Vb, Vc`
A	01_2	00_2	`AND/F Va, Vb, Vc`
A	01_2	01_2	`AND.PN/F Va, Vb, Vc`
A	01_2	10_2	`AND.NP/F Va, Vb, Vc`
A	01_2	11_2	`AND.NN/F Va, Vb, Vc`
C	00_2	00_2	`AND Ra, Rb, #imm`
C	10_2	00_2	`AND Va, Vb, #imm`

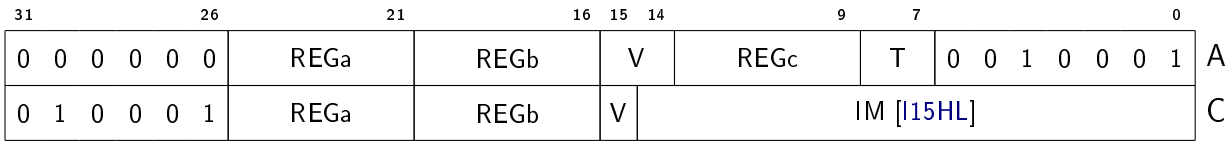
7.6.2 OR - Bitwise or

Compute the bitwise or of two integer operands, with optional negation of the source operands.

Operation

```
if T = 00₂ then
  a ← b | c
else if T = 01₂ then  // .PN
  a ← b | ~c
else if T = 10₂ then  // .NP
  a ← ~b | c
else if T = 11₂ then  // .NN
  a ← ~b | ~c
```

Encoding



Variants

Fmt	V	T	Assembler
A	00_2	00_2	`OR Ra, Rb, Rc`
A	00_2	01_2	`OR.PN Ra, Rb, Rc`
A	00_2	10_2	`OR.NP Ra, Rb, Rc`
A	00_2	11_2	`OR.NN Ra, Rb, Rc`
A	10_2	00_2	`OR Va, Vb, Rc`
A	10_2	01_2	`OR.PN Va, Vb, Rc`
A	10_2	10_2	`OR.NP Va, Vb, Rc`
A	10_2	11_2	`OR.NN Va, Vb, Rc`
A	11_2	00_2	`OR Va, Vb, Vc`
A	11_2	01_2	`OR.PN Va, Vb, Vc`
A	11_2	10_2	`OR.NP Va, Vb, Vc`
A	11_2	11_2	`OR.NN Va, Vb, Vc`
A	01_2	00_2	`OR/F Va, Vb, Vc`
A	01_2	01_2	`OR.PN/F Va, Vb, Vc`
A	01_2	10_2	`OR.NP/F Va, Vb, Vc`
A	01_2	11_2	`OR.NN/F Va, Vb, Vc`
C	00_2	00_2	`OR Ra, Rb, #imm`
C	10_2	00_2	`OR Va, Vb, #imm`

7.6.3 XOR - Bitwise exclusive or

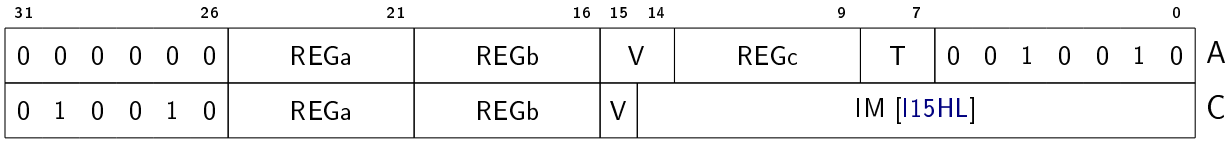
Compute the bitwise exclusive or of two integer operands, with optional negation of the source operands.

Operation

```
if T = 00₂ then
  a ← b ^ c
else if T = 01₂ then  // .PN
  a ← b ^ ~c
```

```
else if T = 10₂ then  // .NP
  a ← ~b ^ c
else if T = 11₂ then  // .NN
  a ← ~b ^ ~c
```

Encoding



Variants

Fmt	V	T	Assembler
A	00_2	00_2	`XOR  Ra, Rb, Rc`
A	00_2	01_2	`XOR.PN  Ra, Rb, Rc`
A	00_2	10_2	`XOR.NP  Ra, Rb, Rc`
A	00_2	11_2	`XOR.NN  Ra, Rb, Rc`
A	10_2	00_2	`XOR  Va, Vb, Rc`
A	10_2	01_2	`XOR.PN  Va, Vb, Rc`
A	10_2	10_2	`XOR.NP  Va, Vb, Rc`
A	10_2	11_2	`XOR.NN  Va, Vb, Rc`
A	11_2	00_2	`XOR  Va, Vb, Vc`
A	11_2	01_2	`XOR.PN  Va, Vb, Vc`
A	11_2	10_2	`XOR.NP  Va, Vb, Vc`
A	11_2	11_2	`XOR.NN  Va, Vb, Vc`
A	01_2	00_2	`XOR/F  Va, Vb, Vc`
A	01_2	01_2	`XOR.PN/F Va, Vb, Vc`
A	01_2	10_2	`XOR.NP/F Va, Vb, Vc`
A	01_2	11_2	`XOR.NN/F Va, Vb, Vc`
C	00_2	00_2	`XOR  Ra, Rb, #imm`
C	10_2	00_2	`XOR  Va, Vb, #imm`

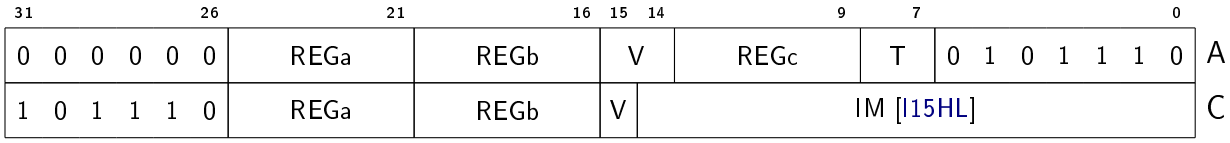
7.6.4 SEL - Bitwise select

Select bits from two different source operands based on the bit values in a third operand, and store the result in the destination operand.

Operation

```
if T = 00₂ then
  a ← (b & a) | (c & ~a)
else if T = 01₂ then      // .132
  a ← (c & a) | (b & ~a)
else if T = 10₂ then      // .213
  a ← (a & b) | (c & ~b)
else if T = 11₂ then      // .231
  a ← (c & b) | (a & ~b)
```

Encoding



Variants

Fmt	V	T	Assembler
A	00_2	00_2	SEL Ra, Rb, Rc
A	00_2	01_2	SEL.132 Ra, Rb, Rc
A	00_2	10_2	SEL.213 Ra, Rb, Rc
A	00_2	11_2	SEL.231 Ra, Rb, Rc
A	10_2	00_2	SEL Va, Vb, Rc
A	10_2	01_2	SEL.132 Va, Vb, Rc
A	10_2	10_2	SEL.213 Va, Vb, Rc
A	10_2	11_2	SEL.231 Va, Vb, Rc
A	11_2	00_2	SEL Va, Vb, Vc
A	11_2	01_2	SEL.132 Va, Vb, Vc
A	11_2	10_2	SEL.213 Va, Vb, Vc
A	11_2	11_2	SEL.231 Va, Vb, Vc
A	01_2	00_2	SEL/F Va, Vb, Vc
A	01_2	01_2	SEL.132/F Va, Vb, Vc
A	01_2	10_2	SEL.213/F Va, Vb, Vc
A	01_2	11_2	SEL.231/F Va, Vb, Vc
C	00_2	00_2	SEL Ra, Rb, #imm
C	10_2	00_2	SEL Va, Vb, #imm

Note

The operation involves four operands (three source operands and one destination operand). However, since the instruction encoding format only allows for three operands in total, one of the source operands (a) is also the destination operand.

For increased flexibility, there are several variants of this instruction that use different permutations of the source operands, which makes it possible to select which of the source registers to clobber.

The numbers in the permutation suffix (.132, .213 or .231) idicate the operand position of the selector operand (first number), the if-one operand (second number), and the if-zero operand (third number). When no suffix is given, the operand position order is 123.

For instance, SEL.231 R6,R7,R8 implements R6 = (R8 & R7) | (R6 & ~R7).

7.7 Bit manipulation

7.7.1 EBF - Extract bit field

Extract a bit field from the first source operand, sign-extend it, and store it in the destination operand. The bit field (offset, width) is defined by the second source operand.

In word mode, bits <12:8> of the second source operand describe the bit field width, and bits <4:0> describe the bit field offset.

In half-word mode, bits <11:8> of the second source operand describe the bit field width, and bits <3:0> describe the bit field offset.

In byte mode, bits <6:4> of the second source operand describe the bit field width, and bits <2:0> describe the bit field offset.

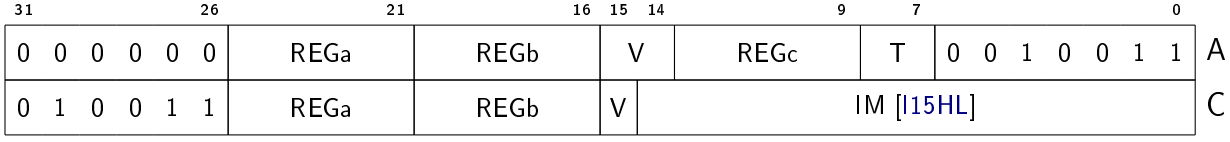
If the value of the bit field width descriptor is zero (0), the width is interpreted as being the full width of the slice.

The first source operand is sign extended up to the number of bits required by the bit field.

Operation

```
o ← uint(c & (bits-1))
if bits = 8 then
  w ← uint((c >> 4) & (bits-1))
else
  w ← uint((c >> 8) & (bits-1))
if w = 0 then
  w ← bits
// Note: b is sign-extended up to o+w bits if o+w > bits.
a ← int(b<o+w-1:o>)
```

Encoding



Variants

Fmt	V	T	Assembler
A	00_2	00_2	`EBF Ra, Rb, Rc`
A	00_2	01_2	`EBF.B Ra, Rb, Rc`
A	00_2	10_2	`EBF.H Ra, Rb, Rc`
A	10_2	00_2	`EBF Va, Vb, Rc`
A	10_2	01_2	`EBF.B Va, Vb, Rc`
A	10_2	10_2	`EBF.H Va, Vb, Rc`
A	11_2	00_2	`EBF Va, Vb, Vc`
A	11_2	01_2	`EBF.B Va, Vb, Vc`
A	11_2	10_2	`EBF.H Va, Vb, Vc`
A	01_2	00_2	`EBF/F Va, Vb, Vc`
A	01_2	01_2	`EBF.B/F Va, Vb, Vc`
A	01_2	10_2	`EBF.H/F Va, Vb, Vc`
C	00_2	00_2	`EBF Ra, Rb, #<offs:width>`
C	10_2	00_2	`EBF Va, Vb, #<offs:width>`

> **Note**
>
> When the bit field width descriptor is all zeros (specifying the full width of the slice), this instruction operates as an arithmetic shift right instruction.
>
> ASR is a valid assembler alias for EBF.

7.7.2 EBFU - Extract bit field unsigned

Extract a bit field from the first source operand, zero-extend it, and store it in the destination operand. The bit field (offset, width) is defined by the second source operand.

In word mode, bits <12:8> of the second source operand describe the bit field width, and bits <4:0> describe the bit field offset.

In half-word mode, bits <11:8> of the second source operand describe the bit field width, and bits <3:0> describe the bit field offset.

In byte mode, bits <6:4> of the second source operand describe the bit field width, and bits <2:0> describe the bit field offset.

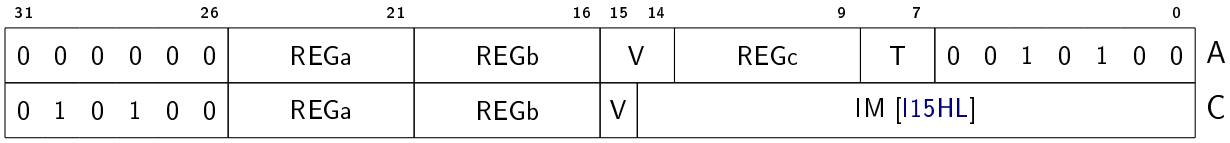
If the value of the bit field width descriptor is zero (0), the width is interpreted as being the full width of the slice.

The first source operand is zero extended up to the number of bits required by the bit field.

Operation

```
o ← uint(c & (bits-1))
if bits = 8 then
  w ← uint((c >> 4) & (bits-1))
else
  w ← uint((c >> 8) & (bits-1))
if w = 0 then
  w ← bits
a ← (b >> o) & ones(w)
```

Encoding



Variants

Fmt	V	T	Assembler
A	00_2	00_2	`EBFU Ra, Rb, Rc`
A	00_2	01_2	`EBFU.B Ra, Rb, Rc`
A	00_2	10_2	`EBFU.H Ra, Rb, Rc`
A	10_2	00_2	`EBFU Va, Vb, Rc`
A	10_2	01_2	`EBFU.B Va, Vb, Rc`
A	10_2	10_2	`EBFU.H Va, Vb, Rc`
A	11_2	00_2	`EBFU Va, Vb, Vc`
A	11_2	01_2	`EBFU.B Va, Vb, Vc`
A	11_2	10_2	`EBFU.H Va, Vb, Vc`
A	01_2	00_2	`EBFU/F Va, Vb, Vc`
A	01_2	01_2	`EBFU.B/F Va, Vb, Vc`
A	01_2	10_2	`EBFU.H/F Va, Vb, Vc`
C	00_2	00_2	`EBFU Ra, Rb, #<offs:width>`
C	10_2	00_2	`EBFU Va, Vb, #<offs:width>`

Note

When the bit field width descriptor is all zeros (specifying the full width of the slice), this instruction operates as a logic shift right instruction.

LSR is a valid assembler alias for EBFU.

7.7.3 MKBF - Make bit field

Extract a bit field from the lower bits of the first source operand, shift it to the left, and store it in the destination operand.

The bit field (offset, width) is defined by the second source operand.

In word mode, bits <12:8> of the second source operand describe the bit field width, and bits <4:0> describe the bit field offset.

In half-word mode, bits <11:8> of the second source operand describe the bit field width, and bits <3:0> describe the bit field offset.

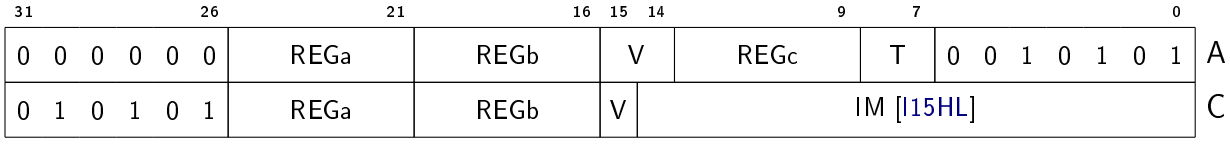
In byte mode, bits <6:4> of the second source operand describe the bit field width, and bits <2:0> describe the bit field offset.

If the value of the bit field width descriptor is zero (0), the width is interpreted as being the full width of the slice.

Operation

```
o ← uint(c & (bits-1))
if bits = 8 then
  w ← uint((c >> 4) & (bits-1))
else
  w ← uint((c >> 8) & (bits-1))
if w = 0 then
  w ← bits
a ← (b & ones(w)) << o
```

Encoding



Variants

Fmt	V	T	Assembler
A	00_2	00_2	`MKBF Ra, Rb, Rc`
A	00_2	01_2	`MKBF.B Ra, Rb, Rc`
A	00_2	10_2	`MKBF.H Ra, Rb, Rc`
A	10_2	00_2	`MKBF Va, Vb, Rc`
A	10_2	01_2	`MKBF.B Va, Vb, Rc`
A	10_2	10_2	`MKBF.H Va, Vb, Rc`
A	11_2	00_2	`MKBF Va, Vb, Vc`
A	11_2	01_2	`MKBF.B Va, Vb, Vc`
A	11_2	10_2	`MKBF.H Va, Vb, Vc`
A	01_2	00_2	`MKBF/F Va, Vb, Vc`
A	01_2	01_2	`MKBF.B/F Va, Vb, Vc`
A	01_2	10_2	`MKBF.H/F Va, Vb, Vc`
C	00_2	00_2	`MKBF Ra, Rb, #<offs:width>`
C	10_2	00_2	`MKBF Va, Vb, #<offs:width>`

> **Note**
>
> When the bit field width descriptor is all zeros (specifying the full width of the slice), this instruction operates as a logic shift left instruction.
>
> LSL is a valid assembler alias for MKBF.

7.7.4 IBF - Insert bit field

Extract a bit field from the lower bits of the first source operand, shift it to the left, and insert it in the destination operand.

The bit field (offset, width) is defined by the second source operand.

In word mode, bits <12:8> of the second source operand describe the bit field width, and bits <4:0> describe the bit field offset.

In half-word mode, bits <11:8> of the second source operand describe the bit field width, and bits <3:0> describe the bit field offset.

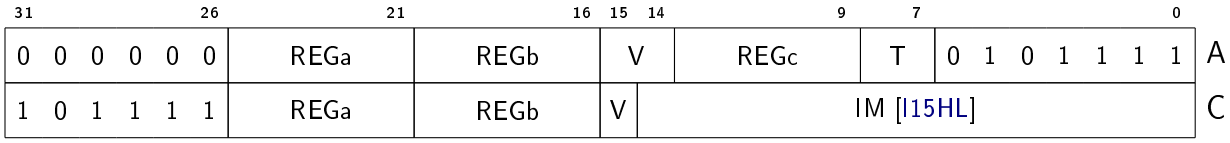
In byte mode, bits <6:4> of the second source operand describe the bit field width, and bits <2:0> describe the bit field offset.

If the value of the bit field width descriptor is zero (0), the width is interpreted as being the full width of the slice.

Operation

```
o ← uint(c & (bits-1))
if bits = 8 then
  w ← uint((c >> 4) & (bits-1))
else
  w ← uint((c >> 8) & (bits-1))
if w = 0 then
  w ← bits
a ← (a & ~(ones(w) << o)) | ((b & ones(w)) << o)
```

Encoding



Variants

Fmt	V	T	Assembler
A	00_2	00_2	IBF Ra, Rb, Rc
A	00_2	01_2	IBF.B Ra, Rb, Rc
A	00_2	10_2	IBF.H Ra, Rb, Rc
A	10_2	00_2	IBF Va, Vb, Rc
A	10_2	01_2	IBF.B Va, Vb, Rc
A	10_2	10_2	IBF.H Va, Vb, Rc
A	11_2	00_2	IBF Va, Vb, Vc
A	11_2	01_2	IBF.B Va, Vb, Vc
A	11_2	10_2	IBF.H Va, Vb, Vc
A	01_2	00_2	IBF/F Va, Vb, Vc
A	01_2	01_2	IBF.B/F Va, Vb, Vc
A	01_2	10_2	IBF.H/F Va, Vb, Vc
C	00_2	00_2	IBF Ra, Rb, #<offs:width>
C	10_2	00_2	IBF Va, Vb, #<offs:width>

7.7.5 SHUF - Shuffle bytes

Shuffle bytes with optional zero- or sign-extension.

The bytes of the second operand are shuffled according to the 13-bit control word in the third operand, and the result is stored in the the first operand.

Bit 12 of the control word determines the sign mode: 1 = sign fill, 0 = zero fill.

Bits 0-1 give the source byte index for destination byte 0, and bit 2 gives the fill mode (0 = verbatim copy, 1 = fill).

Likewise, bits 3-5 define destination byte 1, bits 6-8 define destination byte 2 and bits 9-11 define destination byte 3.

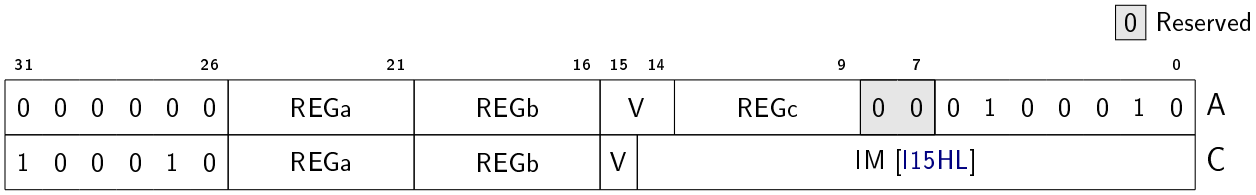
When the fill mode is 0, the source byte is copied to the destination byte. When the fill mode is 1, the destination byte is filled with the most significant bit of the source byte if the sign mode is 1, or zeros if the sign mode is 0.

Note: Byte 0 is the least significant byte (bits 0-7) and byte 3 is the most significant byte (bits 24-31).

Operation

```
sign ← c<12>
for k in 0 to 3
  fill ← c<k×3+2>
  idx ← uint(c<k×3+1:k×3>)
  byte ← b<8×idx+7:8×idx>
  if fill = 1 then
    if sign & byte<7> = 1 then
      byte ← ones(8)
    else
      byte ← zeros(8)
  a<8×k+7:8×k> ← byte
```

Encoding



Variants

Fmt	V	Assembler
A	00_2	`SHUF Ra, Rb, Rc`
A	10_2	`SHUF Va, Vb, Rc`
A	11_2	`SHUF Va, Vb, Vc`
A	01_2	`SHUF/F Va, Vb, Vc`
C	00_2	`SHUF Ra, Rb, #imm`
C	10_2	`SHUF Va, Vb, #imm`

> **Note**
>
> The instruction can be used for several different common tasks, such as zero- and sign-extension of integer bytes and half-words, unpacking of byte-aligend fields within a word, applying byte masks (e.g. bitwise and with 0x00ff00ff) and/or changing the byte order (e.g. for conversion between big and little endian formats or different RGBA color formats).

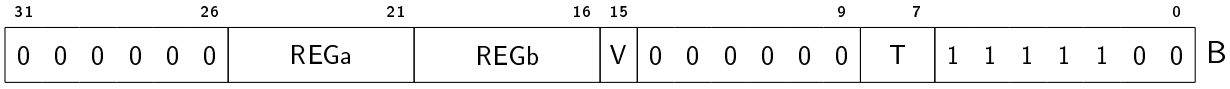
7.7.6 REV - Reverse bits

Reverse the bits of the source operand.

Operation

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

Encoding



Variants

V	T	Assembler	
00_2	00_2	REV	Ra, Rb
00_2	01_2	REV.B	Ra, Rb
00_2	10_2	REV.H	Ra, Rb
10_2	00_2	REV	Va, Vb
10_2	01_2	REV.B	Va, Vb
10_2	10_2	REV.H	Va, Vb

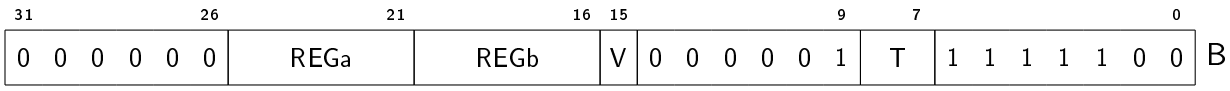
7.7.7 CLZ - Count leading zeros

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

Operation

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

Encoding



Variants

V	T	Assembler	
00_2	00_2	CLZ	Ra, Rb
00_2	01_2	CLZ.B	Ra, Rb
00_2	10_2	CLZ.H	Ra, Rb
10_2	00_2	CLZ	Va, Vb
10_2	01_2	CLZ.B	Va, Vb
10_2	10_2	CLZ.H	Va, Vb

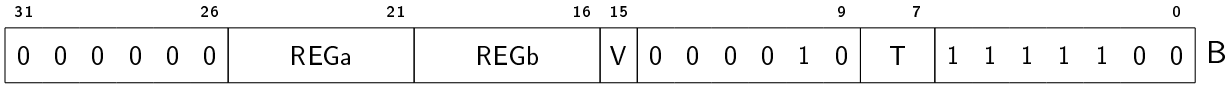
7.7.8 POPCNT - Population count

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

Operation

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

Encoding



Variants

V	T	Assembler
00_2	00_2	`POPCNT  Ra, Rb`
00_2	01_2	`POPCNT.B Ra, Rb`
00_2	10_2	`POPCNT.H Ra, Rb`
10_2	00_2	`POPCNT  Va, Vb`
10_2	01_2	`POPCNT.B Va, Vb`
10_2	10_2	`POPCNT.H Va, Vb`

7.8 Checksum

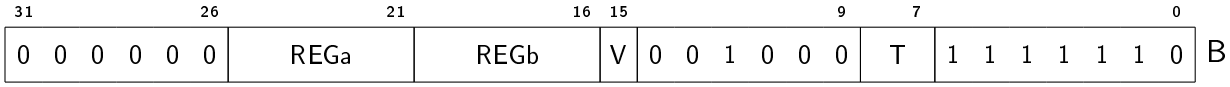
7.8.1 CRC32C - Calculate CRC-32C checksum

Calculate one step of a CRC-32C checksum (polynomial 0x1edc6f41, Castagnoli).

Operation

```
if T = 00₂ then           // .8
  a ← crc32c(a, b)
else if T = 01₂ then      // .16
  a ← crc32c(a, b)
  a ← crc32c(a, b >> 8)
else if T = 10₂ then      // .32
  a ← crc32c(a, b)
  a ← crc32c(a, b >> 8)
  a ← crc32c(a, b >> 16)
  a ← crc32c(a, b >> 24)
```

Encoding



Variants

V	T	Assembler
00_2	00_2	CRC32C.8 Ra, Rb
00_2	01_2	CRC32C.16 Ra, Rb
00_2	10_2	CRC32C.32 Ra, Rb
10_2	00_2	CRC32C.8 Va, Vb
10_2	01_2	CRC32C.16 Va, Vb
10_2	10_2	CRC32C.32 Va, Vb

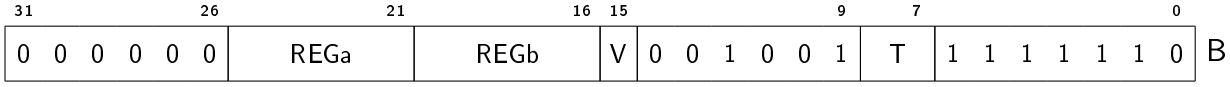
7.8.2 CRC32 - Calculate CRC-32 checksum

Calculate one step of a CRC-32 checksum (polynomial 0x04c11db7).

Operation

```
if T = 00₂ then           // .8
  a ← crc32(a, b)
else if T = 01₂ then      // .16
  a ← crc32(a, b)
  a ← crc32(a, b >> 8)
else if T = 10₂ then      // .32
  a ← crc32(a, b)
  a ← crc32(a, b >> 8)
  a ← crc32(a, b >> 16)
  a ← crc32(a, b >> 24)
```

Encoding



Variants

V	T	Assembler
00_2	00_2	`CRC32.8 Ra, Rb`
00_2	01_2	`CRC32.16 Ra, Rb`
00_2	10_2	`CRC32.32 Ra, Rb`
10_2	00_2	`CRC32.8 Va, Vb`
10_2	01_2	`CRC32.16 Va, Vb`
10_2	10_2	`CRC32.32 Va, Vb`

7.9 Floating-point arithmetic

7.9.1 FADD - Floating-point add

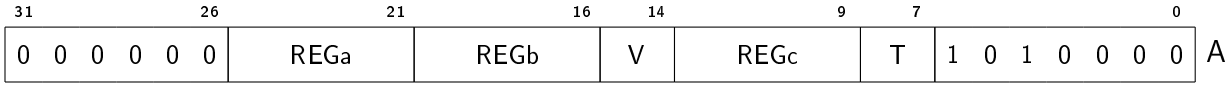
Requires: FM

Compute the sum of two floating-point operands.

Operation

```
a ← float(b) + float(c)
```

Encoding



Variants

V	T	Assembler
00_2	00_2	`FADD Ra, Rb, Rc`
00_2	01_2	`FADD.B Ra, Rb, Rc`
00_2	10_2	`FADD.H Ra, Rb, Rc`
10_2	00_2	`FADD Va, Vb, Rc`
10_2	01_2	`FADD.B Va, Vb, Rc`
10_2	10_2	`FADD.H Va, Vb, Rc`
11_2	00_2	`FADD Va, Vb, Vc`
11_2	01_2	`FADD.B Va, Vb, Vc`
11_2	10_2	`FADD.H Va, Vb, Vc`
01_2	00_2	`FADD/F Va, Vb, Vc`
01_2	01_2	`FADD.B/F Va, Vb, Vc`
01_2	10_2	`FADD.H/F Va, Vb, Vc`

7.9.2 FSUB - Floating-point subtract

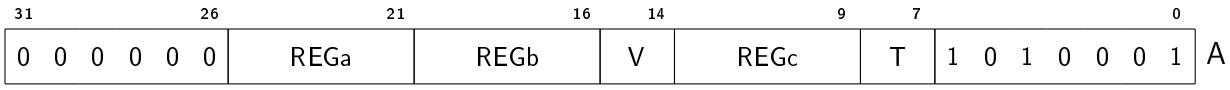
Requires: FM

Compute the difference of two floating-point operands.

Operation

```
a ← float(b) - float(c)
```

Encoding



Variants

V	T	Assembler
00_2	00_2	`FSUB Ra, Rb, Rc`
00_2	01_2	`FSUB.B Ra, Rb, Rc`
00_2	10_2	`FSUB.H Ra, Rb, Rc`
10_2	00_2	`FSUB Va, Vb, Rc`
10_2	01_2	`FSUB.B Va, Vb, Rc`
10_2	10_2	`FSUB.H Va, Vb, Rc`
11_2	00_2	`FSUB Va, Vb, Vc`
11_2	01_2	`FSUB.B Va, Vb, Vc`
11_2	10_2	`FSUB.H Va, Vb, Vc`
01_2	00_2	`FSUB/F Va, Vb, Vc`
01_2	01_2	`FSUB.B/F Va, Vb, Vc`
01_2	10_2	`FSUB.H/F Va, Vb, Vc`

7.9.3 FMUL - Floating-point multiply

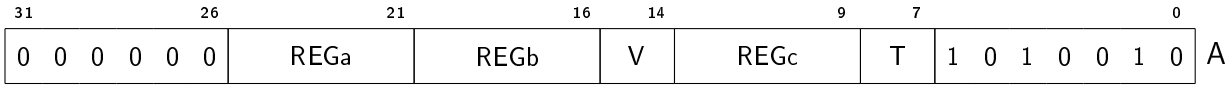
Requires: FM

Compute the product of two floating-point operands.

Operation

```
a ← float(b) × float(c)
```

Encoding



Variants

V	T	Assembler
00_2	00_2	`FMUL Ra, Rb, Rc`
00_2	01_2	`FMUL.B Ra, Rb, Rc`
00_2	10_2	`FMUL.H Ra, Rb, Rc`
10_2	00_2	`FMUL Va, Vb, Rc`
10_2	01_2	`FMUL.B Va, Vb, Rc`
10_2	10_2	`FMUL.H Va, Vb, Rc`
11_2	00_2	`FMUL Va, Vb, Vc`
11_2	01_2	`FMUL.B Va, Vb, Vc`
11_2	10_2	`FMUL.H Va, Vb, Vc`
01_2	00_2	`FMUL/F Va, Vb, Vc`
01_2	01_2	`FMUL.B/F Va, Vb, Vc`
01_2	10_2	`FMUL.H/F Va, Vb, Vc`

7.9.4 FDIV - Floating-point divide

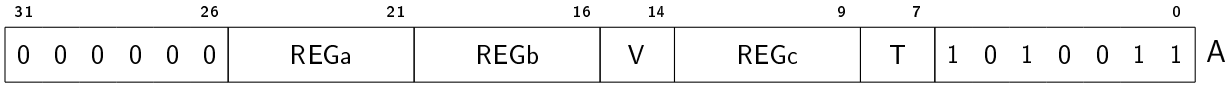
Requires: FM

Compute the quotient of two floating-point operands.

Operation

```
a ← float(b) / float(c)
```

Encoding



Variants

V	T	Assembler
00_2	00_2	`FDIV Ra, Rb, Rc`
00_2	01_2	`FDIV.B Ra, Rb, Rc`
00_2	10_2	`FDIV.H Ra, Rb, Rc`
10_2	00_2	`FDIV Va, Vb, Rc`
10_2	01_2	`FDIV.B Va, Vb, Rc`
10_2	10_2	`FDIV.H Va, Vb, Rc`
11_2	00_2	`FDIV Va, Vb, Vc`
11_2	01_2	`FDIV.B Va, Vb, Vc`
11_2	10_2	`FDIV.H Va, Vb, Vc`
01_2	00_2	`FDIV/F Va, Vb, Vc`
01_2	01_2	`FDIV.B/F Va, Vb, Vc`
01_2	10_2	`FDIV.H/F Va, Vb, Vc`

7.9.5 FMIN - Floating-point minimum

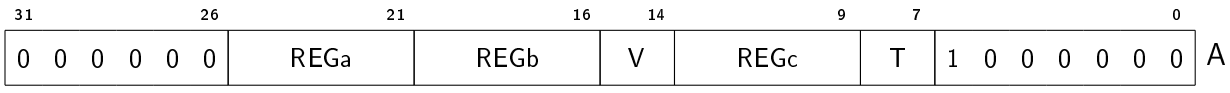
Requires: FM

Return the minimum value of two floating-point operands.

Operation

```
a ← min(float(b), float(c))
```

Encoding



Variants

V	T	Assembler
00_2	00_2	`FMIN Ra, Rb, Rc`
00_2	01_2	`FMIN.B Ra, Rb, Rc`
00_2	10_2	`FMIN.H Ra, Rb, Rc`
10_2	00_2	`FMIN Va, Vb, Rc`
10_2	01_2	`FMIN.B Va, Vb, Rc`
10_2	10_2	`FMIN.H Va, Vb, Rc`
11_2	00_2	`FMIN Va, Vb, Vc`
11_2	01_2	`FMIN.B Va, Vb, Vc`
11_2	10_2	`FMIN.H Va, Vb, Vc`
01_2	00_2	`FMIN/F Va, Vb, Vc`
01_2	01_2	`FMIN.B/F Va, Vb, Vc`
01_2	10_2	`FMIN.H/F Va, Vb, Vc`

7.9.6 FMAX - Floating-point maximum

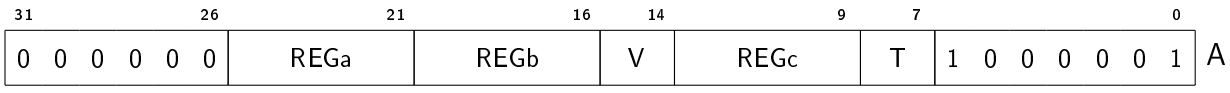
Requires: FM

Return the maximum value of two floating-point operands.

Operation

```
a ← max(float(b), float(c))
```

Encoding



Variants

V	T	Assembler
00_2	00_2	`FMAX Ra, Rb, Rc`
00_2	01_2	`FMAX.B Ra, Rb, Rc`
00_2	10_2	`FMAX.H Ra, Rb, Rc`
10_2	00_2	`FMAX Va, Vb, Rc`
10_2	01_2	`FMAX.B Va, Vb, Rc`
10_2	10_2	`FMAX.H Va, Vb, Rc`
11_2	00_2	`FMAX Va, Vb, Vc`
11_2	01_2	`FMAX.B Va, Vb, Vc`
11_2	10_2	`FMAX.H Va, Vb, Vc`
01_2	00_2	`FMAX/F Va, Vb, Vc`
01_2	01_2	`FMAX.B/F Va, Vb, Vc`
01_2	10_2	`FMAX.H/F Va, Vb, Vc`

7.10 Floating-point comparison

7.10.1 FSEQ - Floating-point set if equal

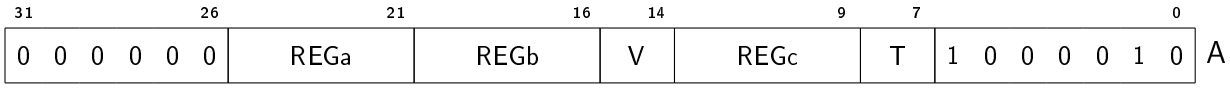
Requires: FM

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.

Operation

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

Encoding



Variants

V	T	Assembler
00_2	00_2	`FSEQ Ra, Rb, Rc`
00_2	01_2	`FSEQ.B Ra, Rb, Rc`
00_2	10_2	`FSEQ.H Ra, Rb, Rc`
10_2	00_2	`FSEQ Va, Vb, Rc`
10_2	01_2	`FSEQ.B Va, Vb, Rc`
10_2	10_2	`FSEQ.H Va, Vb, Rc`
11_2	00_2	`FSEQ Va, Vb, Vc`
11_2	01_2	`FSEQ.B Va, Vb, Vc`
11_2	10_2	`FSEQ.H Va, Vb, Vc`
01_2	00_2	`FSEQ/F Va, Vb, Vc`
01_2	01_2	`FSEQ.B/F Va, Vb, Vc`
01_2	10_2	`FSEQ.H/F Va, Vb, Vc`

7.10.2 FSNE - Floating-point set if not equal

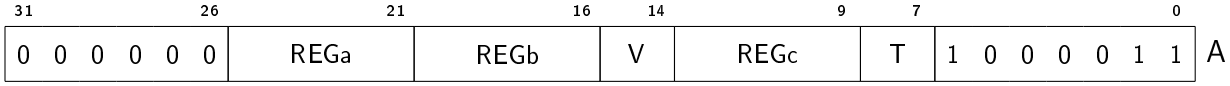
Requires: FM

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.

Operation

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

Encoding



Variants

V	T	Assembler
00_2	00_2	`FSNE Ra, Rb, Rc`
00_2	01_2	`FSNE.B Ra, Rb, Rc`
00_2	10_2	`FSNE.H Ra, Rb, Rc`
10_2	00_2	`FSNE Va, Vb, Rc`
10_2	01_2	`FSNE.B Va, Vb, Rc`
10_2	10_2	`FSNE.H Va, Vb, Rc`
11_2	00_2	`FSNE Va, Vb, Vc`
11_2	01_2	`FSNE.B Va, Vb, Vc`
11_2	10_2	`FSNE.H Va, Vb, Vc`
01_2	00_2	`FSNE/F Va, Vb, Vc`
01_2	01_2	`FSNE.B/F Va, Vb, Vc`
01_2	10_2	`FSNE.H/F Va, Vb, Vc`

7.10.3 FSLT - Floating-point set if less than

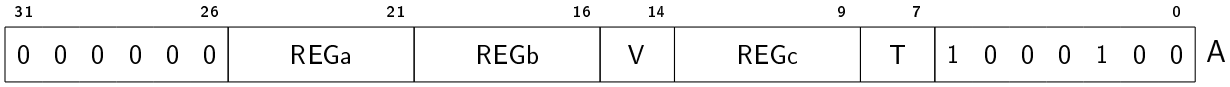
Requires: FM

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.

Operation

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

Encoding



Variants

V	T	Assembler
00_2	00_2	`FSLT Ra, Rb, Rc`
00_2	01_2	`FSLT.B Ra, Rb, Rc`
00_2	10_2	`FSLT.H Ra, Rb, Rc`
10_2	00_2	`FSLT Va, Vb, Rc`
10_2	01_2	`FSLT.B Va, Vb, Rc`
10_2	10_2	`FSLT.H Va, Vb, Rc`
11_2	00_2	`FSLT Va, Vb, Vc`
11_2	01_2	`FSLT.B Va, Vb, Vc`
11_2	10_2	`FSLT.H Va, Vb, Vc`
01_2	00_2	`FSLT/F Va, Vb, Vc`
01_2	01_2	`FSLT.B/F Va, Vb, Vc`
01_2	10_2	`FSLT.H/F Va, Vb, Vc`

7.10.4 FSLE - Floating-point set if less than or equal

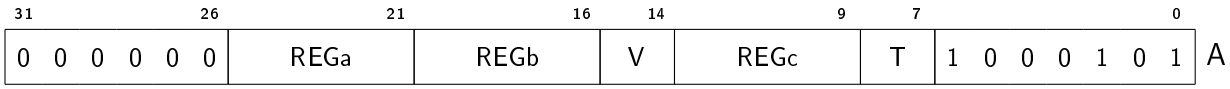
Requires: FM

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.

Operation

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

Encoding



Variants

V	T	Assembler
00_2	00_2	`FSLE Ra, Rb, Rc`
00_2	01_2	`FSLE.B Ra, Rb, Rc`
00_2	10_2	`FSLE.H Ra, Rb, Rc`
10_2	00_2	`FSLE Va, Vb, Rc`
10_2	01_2	`FSLE.B Va, Vb, Rc`
10_2	10_2	`FSLE.H Va, Vb, Rc`
11_2	00_2	`FSLE Va, Vb, Vc`
11_2	01_2	`FSLE.B Va, Vb, Vc`
11_2	10_2	`FSLE.H Va, Vb, Vc`
01_2	00_2	`FSLE/F Va, Vb, Vc`
01_2	01_2	`FSLE.B/F Va, Vb, Vc`
01_2	10_2	`FSLE.H/F Va, Vb, Vc`

7.10.5 FSUNORD - Floating-point set if unordered

Requires: FM

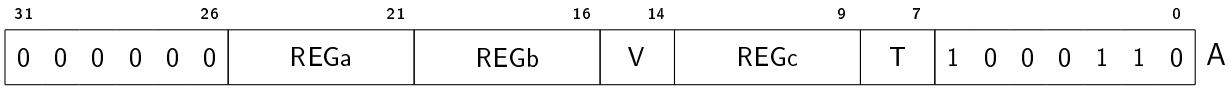
Set all bits of the result to 1 if any of the source operands are undordered (i.e. NaN), otherwise set all bits of the result to 0.

Operation

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

```
else
  a ← zeros(bits)
```

Encoding



Variants

V	T	Assembler
00_2	00_2	`FSUNORD Ra, Rb, Rc`
00_2	01_2	`FSUNORD.B Ra, Rb, Rc`
00_2	10_2	`FSUNORD.H Ra, Rb, Rc`
10_2	00_2	`FSUNORD Va, Vb, Rc`
10_2	01_2	`FSUNORD.B Va, Vb, Rc`
10_2	10_2	`FSUNORD.H Va, Vb, Rc`
11_2	00_2	`FSUNORD Va, Vb, Vc`
11_2	01_2	`FSUNORD.B Va, Vb, Vc`
11_2	10_2	`FSUNORD.H Va, Vb, Vc`
01_2	00_2	`FSUNORD/F Va, Vb, Vc`
01_2	01_2	`FSUNORD.B/F Va, Vb, Vc`
01_2	10_2	`FSUNORD.H/F Va, Vb, Vc`

7.10.6 FSORD - Floating-point set if ordered

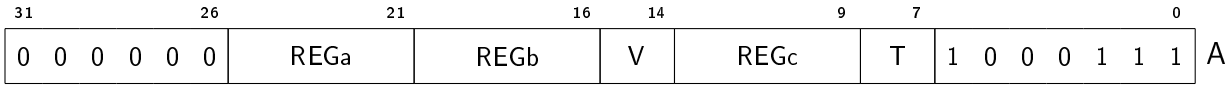
Requires: FM

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.

Operation

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

Encoding



Variants

V	T	Assembler
00_2	00_2	`FSORD Ra, Rb, Rc`
00_2	01_2	`FSORD.B Ra, Rb, Rc`
00_2	10_2	`FSORD.H Ra, Rb, Rc`
10_2	00_2	`FSORD Va, Vb, Rc`
10_2	01_2	`FSORD.B Va, Vb, Rc`
10_2	10_2	`FSORD.H Va, Vb, Rc`
11_2	00_2	`FSORD Va, Vb, Vc`
11_2	01_2	`FSORD.B Va, Vb, Vc`
11_2	10_2	`FSORD.H Va, Vb, Vc`
01_2	00_2	`FSORD/F Va, Vb, Vc`
01_2	01_2	`FSORD.B/F Va, Vb, Vc`
01_2	10_2	`FSORD.H/F Va, Vb, Vc`

7.11 Floating-point conversion

7.11.1 ITOF - Signed integer to floating-point

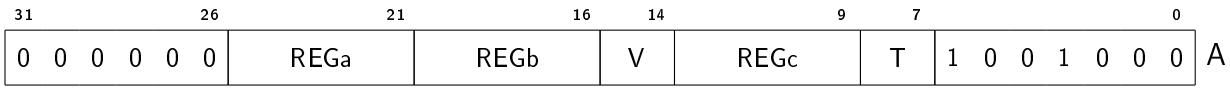
Requires: FM

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.

Operation

```
a ← int2real(int(b)) × pow(2.0, -int(c))
```

Encoding



Variants

V	T	Assembler
00_2	00_2	`ITOF Ra, Rb, Rc`
00_2	01_2	`ITOF.B Ra, Rb, Rc`
00_2	10_2	`ITOF.H Ra, Rb, Rc`
10_2	00_2	`ITOF Va, Vb, Rc`
10_2	01_2	`ITOF.B Va, Vb, Rc`
10_2	10_2	`ITOF.H Va, Vb, Rc`
11_2	00_2	`ITOF Va, Vb, Vc`
11_2	01_2	`ITOF.B Va, Vb, Vc`
11_2	10_2	`ITOF.H Va, Vb, Vc`
01_2	00_2	`ITOF/F Va, Vb, Vc`
01_2	01_2	`ITOF.B/F Va, Vb, Vc`
01_2	10_2	`ITOF.H/F Va, Vb, Vc`

7.11.2 UTOF - Unsigned integer to floating-point

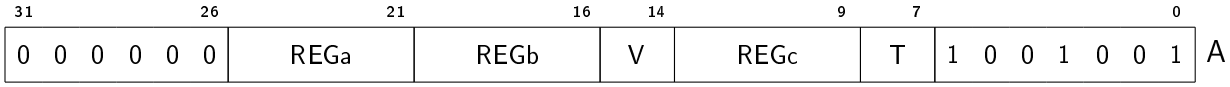
Requires: FM

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.

Operation

```
a ← int2real(uint(b)) × pow(2.0, -int(c))
```

Encoding



Variants

V	T	Assembler
00_2	00_2	`UTOF Ra, Rb, Rc`
00_2	01_2	`UTOF.B Ra, Rb, Rc`
00_2	10_2	`UTOF.H Ra, Rb, Rc`
10_2	00_2	`UTOF Va, Vb, Rc`
10_2	01_2	`UTOF.B Va, Vb, Rc`
10_2	10_2	`UTOF.H Va, Vb, Rc`
11_2	00_2	`UTOF Va, Vb, Vc`
11_2	01_2	`UTOF.B Va, Vb, Vc`
11_2	10_2	`UTOF.H Va, Vb, Vc`
01_2	00_2	`UTOF/F Va, Vb, Vc`
01_2	01_2	`UTOF.B/F Va, Vb, Vc`
01_2	10_2	`UTOF.H/F Va, Vb, Vc`

7.11.3 FTOI - Floating-point to signed integer

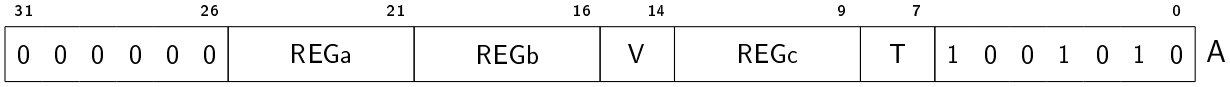
Requires: FM

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.

Operation

```
a ← sat(trunc(float(b) × pow(2.0, int(c))), bits)
```

Encoding



Variants

V	T	Assembler
00_2	00_2	`FTOI Ra, Rb, Rc`
00_2	01_2	`FTOI.B Ra, Rb, Rc`
00_2	10_2	`FTOI.H Ra, Rb, Rc`
10_2	00_2	`FTOI Va, Vb, Rc`
10_2	01_2	`FTOI.B Va, Vb, Rc`
10_2	10_2	`FTOI.H Va, Vb, Rc`
11_2	00_2	`FTOI Va, Vb, Vc`
11_2	01_2	`FTOI.B Va, Vb, Vc`
11_2	10_2	`FTOI.H Va, Vb, Vc`
01_2	00_2	`FTOI/F Va, Vb, Vc`
01_2	01_2	`FTOI.B/F Va, Vb, Vc`
01_2	10_2	`FTOI.H/F Va, Vb, Vc`

7.11.4 FTOU - Floating-point to unsigned integer

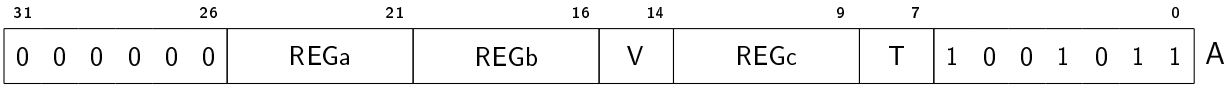
Requires: FM

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.

Operation

```
a ← satu(trunc(float(b) × pow(2.0, int(c))), bits)
```

Encoding



Variants

V	T	Assembler
00_2	00_2	`FTOU Ra, Rb, Rc`
00_2	01_2	`FTOU.B Ra, Rb, Rc`
00_2	10_2	`FTOU.H Ra, Rb, Rc`
10_2	00_2	`FTOU Va, Vb, Rc`
10_2	01_2	`FTOU.B Va, Vb, Rc`
10_2	10_2	`FTOU.H Va, Vb, Rc`
11_2	00_2	`FTOU Va, Vb, Vc`
11_2	01_2	`FTOU.B Va, Vb, Vc`
11_2	10_2	`FTOU.H Va, Vb, Vc`
01_2	00_2	`FTOU/F Va, Vb, Vc`
01_2	01_2	`FTOU.B/F Va, Vb, Vc`
01_2	10_2	`FTOU.H/F Va, Vb, Vc`

7.11.5 FTOIR - Floating-point to signed integer with rounding

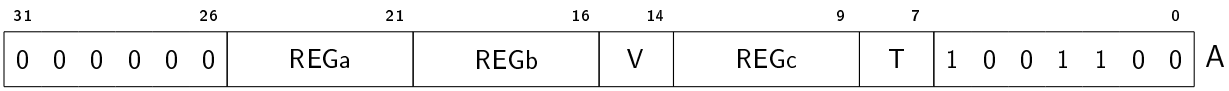
Requires: FM

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.

Operation

```
a ← sat(round(float(b) × pow(2.0, int(c))), bits)
```

Encoding



Variants

V	T	Assembler
00_2	00_2	`FTOIR   Ra, Rb, Rc`
00_2	01_2	`FTOIR.B Ra, Rb, Rc`
00_2	10_2	`FTOIR.H Ra, Rb, Rc`
10_2	00_2	`FTOIR   Va, Vb, Rc`
10_2	01_2	`FTOIR.B Va, Vb, Rc`
10_2	10_2	`FTOIR.H Va, Vb, Rc`
11_2	00_2	`FTOIR   Va, Vb, Vc`
11_2	01_2	`FTOIR.B Va, Vb, Vc`
11_2	10_2	`FTOIR.H Va, Vb, Vc`
01_2	00_2	`FTOIR/F Va, Vb, Vc`
01_2	01_2	`FTOIR.B/F Va, Vb, Vc`
01_2	10_2	`FTOIR.H/F Va, Vb, Vc`

7.11.6 FTOUR - Floating-point to unsigned integer with rounding

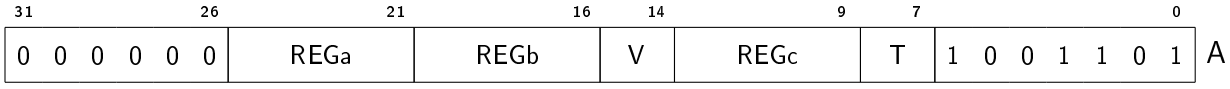
Requires: FM

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.

Operation

```
a ← satu(round(float(b) × pow(2.0, int(c))), bits)
```

Encoding



Variants

V	T	Assembler
00_2	00_2	`FTOUR   Ra, Rb, Rc`
00_2	01_2	`FTOUR.B Ra, Rb, Rc`
00_2	10_2	`FTOUR.H Ra, Rb, Rc`
10_2	00_2	`FTOUR   Va, Vb, Rc`
10_2	01_2	`FTOUR.B Va, Vb, Rc`
10_2	10_2	`FTOUR.H Va, Vb, Rc`
11_2	00_2	`FTOUR   Va, Vb, Vc`
11_2	01_2	`FTOUR.B Va, Vb, Vc`
11_2	10_2	`FTOUR.H Va, Vb, Vc`
01_2	00_2	`FTOUR/F Va, Vb, Vc`
01_2	01_2	`FTOUR.B/F Va, Vb, Vc`
01_2	10_2	`FTOUR.H/F Va, Vb, Vc`

7.12 Packing and unpacking

7.12.1 PACK - Pack

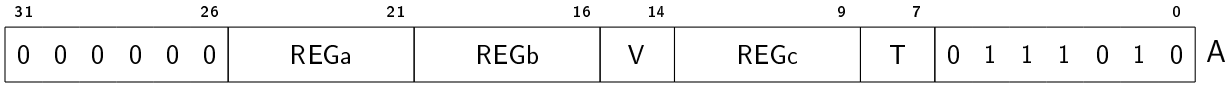
Requires: PM

Pack the lower parts of two integer operands.

Operation

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

Encoding



Variants

V	T	Assembler
00_2	00_2	`PACK Ra, Rb, Rc`
00_2	01_2	`PACK.B Ra, Rb, Rc`
00_2	10_2	`PACK.H Ra, Rb, Rc`
10_2	00_2	`PACK Va, Vb, Rc`
10_2	01_2	`PACK.B Va, Vb, Rc`
10_2	10_2	`PACK.H Va, Vb, Rc`
11_2	00_2	`PACK Va, Vb, Vc`
11_2	01_2	`PACK.B Va, Vb, Vc`
11_2	10_2	`PACK.H Va, Vb, Vc`
01_2	00_2	`PACK/F Va, Vb, Vc`
01_2	01_2	`PACK.B/F Va, Vb, Vc`
01_2	10_2	`PACK.H/F Va, Vb, Vc`

7.12.2 PACKHI - Pack high

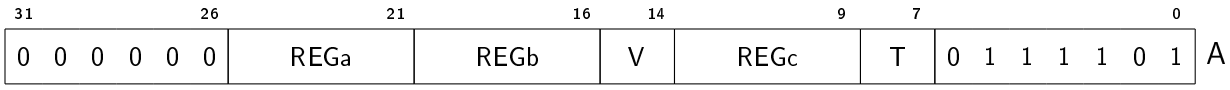
Requires: PM

Pack the higher parts of two integer operands.

Operation

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

Encoding



Variants

V	T	Assembler
00_2	00_2	`PACKHI  Ra, Rb, Rc`
00_2	01_2	`PACKHI.B Ra, Rb, Rc`
00_2	10_2	`PACKHI.H Ra, Rb, Rc`
10_2	00_2	`PACKHI  Va, Vb, Rc`
10_2	01_2	`PACKHI.B Va, Vb, Rc`
10_2	10_2	`PACKHI.H Va, Vb, Rc`
11_2	00_2	`PACKHI  Va, Vb, Vc`
11_2	01_2	`PACKHI.B Va, Vb, Vc`
11_2	10_2	`PACKHI.H Va, Vb, Vc`
01_2	00_2	`PACKHI/F Va, Vb, Vc`
01_2	01_2	`PACKHI.B/F Va, Vb, Vc`
01_2	10_2	`PACKHI.H/F Va, Vb, Vc`

7.12.3 PACKS - Signed pack with saturation

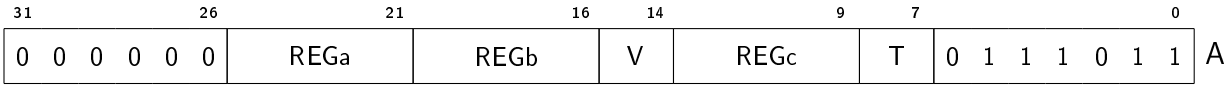
Requires: SM, PM

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

Operation

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

Encoding



Variants

V	T	Assembler
00_2	00_2	`PACKS Ra, Rb, Rc`
00_2	01_2	`PACKS.B Ra, Rb, Rc`
00_2	10_2	`PACKS.H Ra, Rb, Rc`
10_2	00_2	`PACKS Va, Vb, Rc`
10_2	01_2	`PACKS.B Va, Vb, Rc`
10_2	10_2	`PACKS.H Va, Vb, Rc`
11_2	00_2	`PACKS Va, Vb, Vc`
11_2	01_2	`PACKS.B Va, Vb, Vc`
11_2	10_2	`PACKS.H Va, Vb, Vc`
01_2	00_2	`PACKS/F Va, Vb, Vc`
01_2	01_2	`PACKS.B/F Va, Vb, Vc`
01_2	10_2	`PACKS.H/F Va, Vb, Vc`

7.12.4 PACKSU - Unsigned pack with saturation

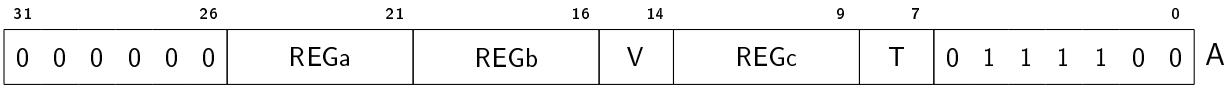
Requires: SM, PM

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

Operation

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

Encoding



Variants

V	T	Assembler
00_2	00_2	`PACKSU  Ra, Rb, Rc`
00_2	01_2	`PACKSU.B Ra, Rb, Rc`
00_2	10_2	`PACKSU.H Ra, Rb, Rc`
10_2	00_2	`PACKSU  Va, Vb, Rc`
10_2	01_2	`PACKSU.B Va, Vb, Rc`
10_2	10_2	`PACKSU.H Va, Vb, Rc`
11_2	00_2	`PACKSU  Va, Vb, Vc`
11_2	01_2	`PACKSU.B Va, Vb, Vc`
11_2	10_2	`PACKSU.H Va, Vb, Vc`
01_2	00_2	`PACKSU/F Va, Vb, Vc`
01_2	01_2	`PACKSU.B/F Va, Vb, Vc`
01_2	10_2	`PACKSU.H/F Va, Vb, Vc`

7.12.5 PACKHIR - Signed pack high with rounding

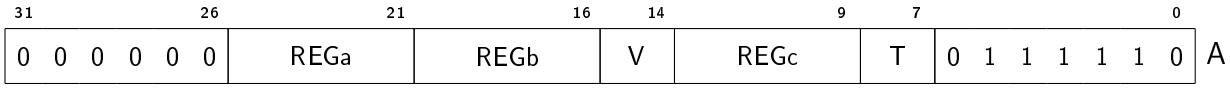
Requires: SM, PM

Round and pack the higher parts of two signed integer operands.

Operation

```
hi ← sat(int(b) + 1<<(bits/2-1), bits)
lo ← sat(int(c) + 1<<(bits/2-1), bits)
a ← (hi & ~ones(bits/2)) | (lo >> (bits/2))
```

Encoding



Variants

V	T	Assembler
00_2	00_2	`PACKHIR Ra, Rb, Rc`
00_2	01_2	`PACKHIR.B Ra, Rb, Rc`
00_2	10_2	`PACKHIR.H Ra, Rb, Rc`
10_2	00_2	`PACKHIR Va, Vb, Rc`
10_2	01_2	`PACKHIR.B Va, Vb, Rc`
10_2	10_2	`PACKHIR.H Va, Vb, Rc`
11_2	00_2	`PACKHIR Va, Vb, Vc`
11_2	01_2	`PACKHIR.B Va, Vb, Vc`
11_2	10_2	`PACKHIR.H Va, Vb, Vc`
01_2	00_2	`PACKHIR/F Va, Vb, Vc`
01_2	01_2	`PACKHIR.B/F Va, Vb, Vc`
01_2	10_2	`PACKHIR.H/F Va, Vb, Vc`

7.12.6 PACKHIUR - Unsigned pack high with rounding

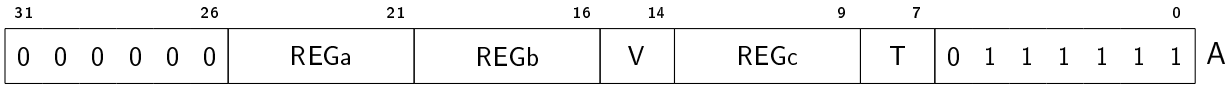
Requires: SM, PM

Round and pack the higher parts of two unsigned integer operands.

Operation

```
hi ← satu(uint(b) + 1<<(bits/2-1), bits)
lo ← satu(uint(c) + 1<<(bits/2-1), bits)
a ← (hi & ~ones(bits/2)) | (lo >> (bits/2))
```

Encoding



Variants

V	T	Assembler
00_2	00_2	`PACKHIUR Ra, Rb, Rc`
00_2	01_2	`PACKHIUR.B Ra, Rb, Rc`
00_2	10_2	`PACKHIUR.H Ra, Rb, Rc`
10_2	00_2	`PACKHIUR Va, Vb, Rc`
10_2	01_2	`PACKHIUR.B Va, Vb, Rc`
10_2	10_2	`PACKHIUR.H Va, Vb, Rc`
11_2	00_2	`PACKHIUR Va, Vb, Vc`
11_2	01_2	`PACKHIUR.B Va, Vb, Vc`
11_2	10_2	`PACKHIUR.H Va, Vb, Vc`
01_2	00_2	`PACKHIUR/F Va, Vb, Vc`
01_2	01_2	`PACKHIUR.B/F Va, Vb, Vc`
01_2	10_2	`PACKHIUR.H/F Va, Vb, Vc`

7.12.7 FPACK - Floating-point pack

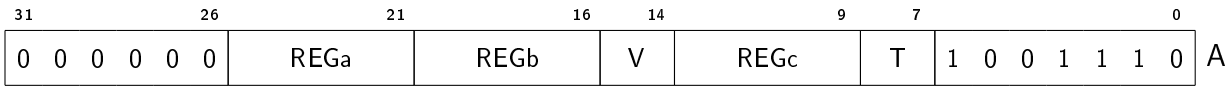
Requires: FM, PM

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.*

Encoding



Variants

V	T	Assembler
00_2	00_2	`FPACK Ra, Rb, Rc`
00_2	10_2	`FPACK.H Ra, Rb, Rc`
10_2	00_2	`FPACK Va, Vb, Rc`
10_2	10_2	`FPACK.H Va, Vb, Rc`
11_2	00_2	`FPACK Va, Vb, Vc`
11_2	10_2	`FPACK.H Va, Vb, Vc`
01_2	00_2	`FPACK/F Va, Vb, Vc`
01_2	10_2	`FPACK.H/F Va, Vb, Vc`

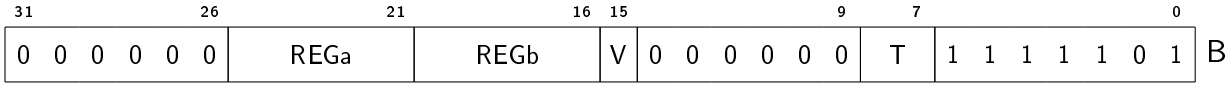
7.12.8 FUNPL - Floating-point unpack low

Requires: FM, PM

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.*

Encoding



Variants

V	T	Assembler
00_2	00_2	`FUNPL Ra, Rb`
00_2	10_2	`FUNPL.H Ra, Rb`
10_2	00_2	`FUNPL Va, Vb`
10_2	10_2	`FUNPL.H Va, Vb`

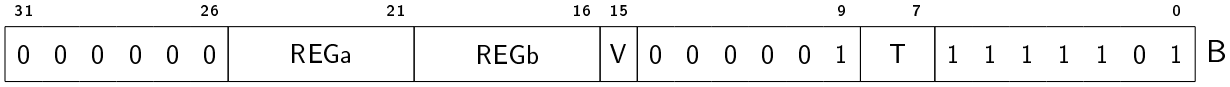
7.12.9 FUNPH - Floating-point unpack high

Requires: FM, PM

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.*

Encoding



Variants

V	T	Assembler
00_2	00_2	`FUNPH  Ra, Rb`
00_2	10_2	`FUNPH.H Ra, Rb`
10_2	00_2	`FUNPH  Va, Vb`
10_2	10_2	`FUNPH.H Va, Vb`

7.13 Saturating and halving arithmetic

7.13.1 ADDS - Signed add with saturation

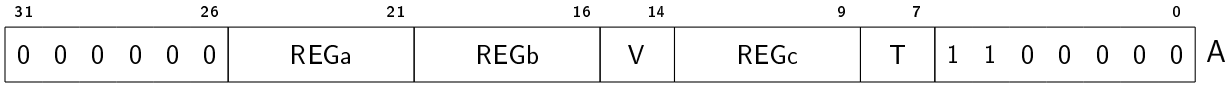
Requires: SM

Compute the saturated sum of two signed integer operands.

Operation

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

Encoding



Variants

V	T	Assembler
00_2	00_2	`ADDS Ra, Rb, Rc`
00_2	01_2	`ADDS.B Ra, Rb, Rc`
00_2	10_2	`ADDS.H Ra, Rb, Rc`
10_2	00_2	`ADDS Va, Vb, Rc`
10_2	01_2	`ADDS.B Va, Vb, Rc`
10_2	10_2	`ADDS.H Va, Vb, Rc`
11_2	00_2	`ADDS Va, Vb, Vc`
11_2	01_2	`ADDS.B Va, Vb, Vc`
11_2	10_2	`ADDS.H Va, Vb, Vc`
01_2	00_2	`ADDS/F Va, Vb, Vc`
01_2	01_2	`ADDS.B/F Va, Vb, Vc`
01_2	10_2	`ADDS.H/F Va, Vb, Vc`

7.13.2 ADDSU - Unsigned add with saturation

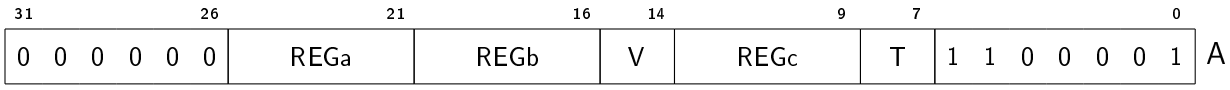
Requires: SM

Compute the saturated sum of two unsigned integer operands.

Operation

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

Encoding



Variants

V	T	Assembler
00_2	00_2	`ADDSU Ra, Rb, Rc`
00_2	01_2	`ADDSU.B Ra, Rb, Rc`
00_2	10_2	`ADDSU.H Ra, Rb, Rc`
10_2	00_2	`ADDSU Va, Vb, Rc`
10_2	01_2	`ADDSU.B Va, Vb, Rc`
10_2	10_2	`ADDSU.H Va, Vb, Rc`
11_2	00_2	`ADDSU Va, Vb, Vc`
11_2	01_2	`ADDSU.B Va, Vb, Vc`
11_2	10_2	`ADDSU.H Va, Vb, Vc`
01_2	00_2	`ADDSU/F Va, Vb, Vc`
01_2	01_2	`ADDSU.B/F Va, Vb, Vc`
01_2	10_2	`ADDSU.H/F Va, Vb, Vc`

7.13.3 ADDH - Signed half add

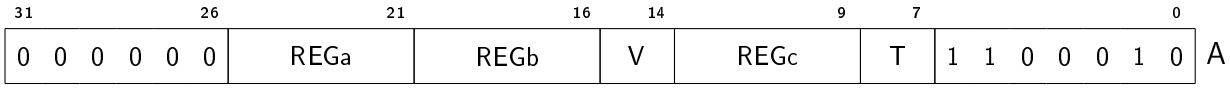
Requires: SM

Compute the half sum of two signed integer operands.

Operation

```
a ← (int(b) + int(c)) >>s 1
```

Encoding



Variants

V	T	Assembler
00_2	00_2	`ADDH Ra, Rb, Rc`
00_2	01_2	`ADDH.B Ra, Rb, Rc`
00_2	10_2	`ADDH.H Ra, Rb, Rc`
10_2	00_2	`ADDH Va, Vb, Rc`
10_2	01_2	`ADDH.B Va, Vb, Rc`
10_2	10_2	`ADDH.H Va, Vb, Rc`
11_2	00_2	`ADDH Va, Vb, Vc`
11_2	01_2	`ADDH.B Va, Vb, Vc`
11_2	10_2	`ADDH.H Va, Vb, Vc`
01_2	00_2	`ADDH/F Va, Vb, Vc`
01_2	01_2	`ADDH.B/F Va, Vb, Vc`
01_2	10_2	`ADDH.H/F Va, Vb, Vc`

7.13.4 ADDHU - Unsigned half add

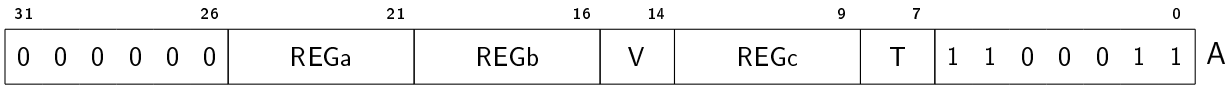
Requires: SM

Compute the half sum of two unsigned integer operands.

Operation

```
a ← (uint(b) + uint(c)) >> 1
```

Encoding



Variants

V	T	Assembler
00_2	00_2	`ADDHU Ra, Rb, Rc`
00_2	01_2	`ADDHU.B Ra, Rb, Rc`
00_2	10_2	`ADDHU.H Ra, Rb, Rc`
10_2	00_2	`ADDHU Va, Vb, Rc`
10_2	01_2	`ADDHU.B Va, Vb, Rc`
10_2	10_2	`ADDHU.H Va, Vb, Rc`
11_2	00_2	`ADDHU Va, Vb, Vc`
11_2	01_2	`ADDHU.B Va, Vb, Vc`
11_2	10_2	`ADDHU.H Va, Vb, Vc`
01_2	00_2	`ADDHU/F Va, Vb, Vc`
01_2	01_2	`ADDHU.B/F Va, Vb, Vc`
01_2	10_2	`ADDHU.H/F Va, Vb, Vc`

7.13.5 ADDHR - Signed half add with rounding

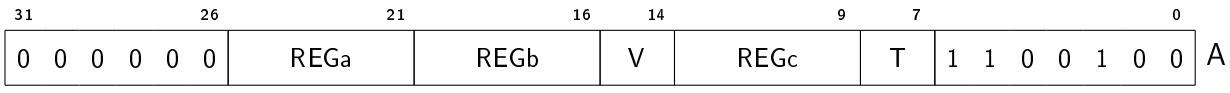
Requires: SM

Compute the rounded half sum of two signed integer operands.

Operation

$a \leftarrow (int(b) + int(c) + 1) >>_s 1$

Encoding



Variants

V	T	Assembler
00_2	00_2	`ADDHR   Ra, Rb, Rc`
00_2	01_2	`ADDHR.B Ra, Rb, Rc`
00_2	10_2	`ADDHR.H Ra, Rb, Rc`
10_2	00_2	`ADDHR   Va, Vb, Rc`
10_2	01_2	`ADDHR.B Va, Vb, Rc`
10_2	10_2	`ADDHR.H Va, Vb, Rc`
11_2	00_2	`ADDHR   Va, Vb, Vc`
11_2	01_2	`ADDHR.B Va, Vb, Vc`
11_2	10_2	`ADDHR.H Va, Vb, Vc`
01_2	00_2	`ADDHR/F Va, Vb, Vc`
01_2	01_2	`ADDHR.B/F Va, Vb, Vc`
01_2	10_2	`ADDHR.H/F Va, Vb, Vc`

7.13.6 ADDHUR - Unsigned half add with rounding

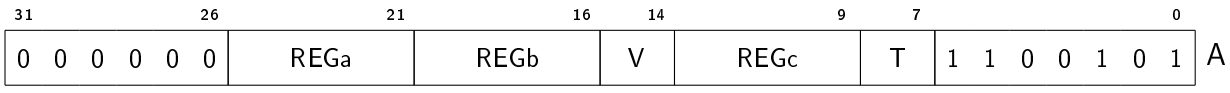
Requires: SM

Compute the runded half sum of two unsigned integer operands.

Operation

```
a ← (uint(b) + uint(c) + 1) >> 1
```

Encoding



Variants

V	T	Assembler
00_2	00_2	`ADDHUR  Ra, Rb, Rc`
00_2	01_2	`ADDHUR.B Ra, Rb, Rc`
00_2	10_2	`ADDHUR.H Ra, Rb, Rc`
10_2	00_2	`ADDHUR  Va, Vb, Rc`
10_2	01_2	`ADDHUR.B Va, Vb, Rc`
10_2	10_2	`ADDHUR.H Va, Vb, Rc`
11_2	00_2	`ADDHUR  Va, Vb, Vc`
11_2	01_2	`ADDHUR.B Va, Vb, Vc`
11_2	10_2	`ADDHUR.H Va, Vb, Vc`
01_2	00_2	`ADDHUR/F Va, Vb, Vc`
01_2	01_2	`ADDHUR.B/F Va, Vb, Vc`
01_2	10_2	`ADDHUR.H/F Va, Vb, Vc`

7.13.7 SUBS - Signed subtract with saturation

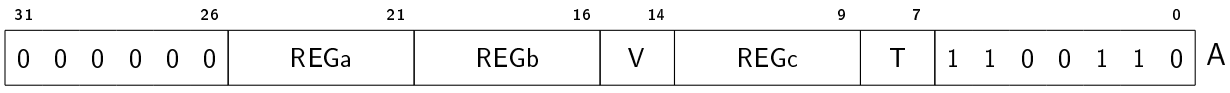
Requires: SM

Compute the saturated difference of two signed integer operands.

Operation

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

Encoding



Variants

V	T	Assembler
00_2	00_2	`SUBS Ra, Rb, Rc`
00_2	01_2	`SUBS.B Ra, Rb, Rc`
00_2	10_2	`SUBS.H Ra, Rb, Rc`
10_2	00_2	`SUBS Va, Vb, Rc`
10_2	01_2	`SUBS.B Va, Vb, Rc`
10_2	10_2	`SUBS.H Va, Vb, Rc`
11_2	00_2	`SUBS Va, Vb, Vc`
11_2	01_2	`SUBS.B Va, Vb, Vc`
11_2	10_2	`SUBS.H Va, Vb, Vc`
01_2	00_2	`SUBS/F Va, Vb, Vc`
01_2	01_2	`SUBS.B/F Va, Vb, Vc`
01_2	10_2	`SUBS.H/F Va, Vb, Vc`

7.13.8 SUBSU - Unsigned subtract with saturation

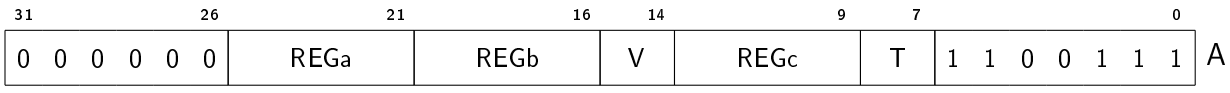
Requires: SM

Compute the saturated difference of two unsigned integer operands.

Operation

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

Encoding



Variants

V	T	Assembler
00_2	00_2	`SUBSU   Ra, Rb, Rc`
00_2	01_2	`SUBSU.B Ra, Rb, Rc`
00_2	10_2	`SUBSU.H Ra, Rb, Rc`
10_2	00_2	`SUBSU   Va, Vb, Rc`
10_2	01_2	`SUBSU.B Va, Vb, Rc`
10_2	10_2	`SUBSU.H Va, Vb, Rc`
11_2	00_2	`SUBSU   Va, Vb, Vc`
11_2	01_2	`SUBSU.B Va, Vb, Vc`
11_2	10_2	`SUBSU.H Va, Vb, Vc`
01_2	00_2	`SUBSU/F Va, Vb, Vc`
01_2	01_2	`SUBSU.B/F Va, Vb, Vc`
01_2	10_2	`SUBSU.H/F Va, Vb, Vc`

7.13.9 SUBH - Signed half subtract

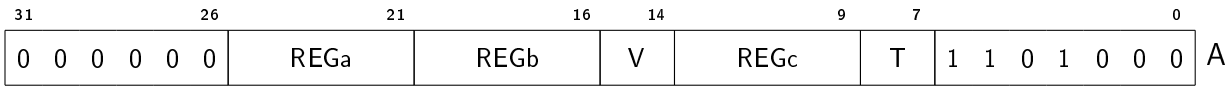
Requires: SM

Compute the half difference of two signed integer operands.

Operation

`a` $\leftarrow$ `(int(b) - int(c))` $>>_s$ `1`

Encoding



Variants

V	T	Assembler
00_2	00_2	`SUBH Ra, Rb, Rc`
00_2	01_2	`SUBH.B Ra, Rb, Rc`
00_2	10_2	`SUBH.H Ra, Rb, Rc`
10_2	00_2	`SUBH Va, Vb, Rc`
10_2	01_2	`SUBH.B Va, Vb, Rc`
10_2	10_2	`SUBH.H Va, Vb, Rc`
11_2	00_2	`SUBH Va, Vb, Vc`
11_2	01_2	`SUBH.B Va, Vb, Vc`
11_2	10_2	`SUBH.H Va, Vb, Vc`
01_2	00_2	`SUBH/F Va, Vb, Vc`
01_2	01_2	`SUBH.B/F Va, Vb, Vc`
01_2	10_2	`SUBH.H/F Va, Vb, Vc`

7.13.10 SUBHU - Unsigned half subtract

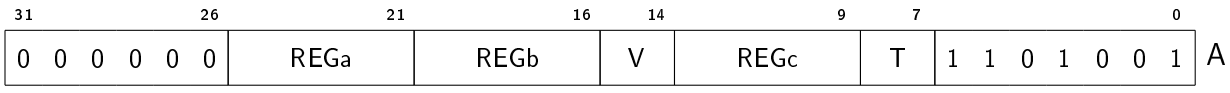
Requires: SM

Compute the half difference of two unsigned integer operands.

Operation

```
a ← (uint(b) - uint(c)) >> 1
```

Encoding



Variants

V	T	Assembler
00_2	00_2	`SUBHU   Ra, Rb, Rc`
00_2	01_2	`SUBHU.B Ra, Rb, Rc`
00_2	10_2	`SUBHU.H Ra, Rb, Rc`
10_2	00_2	`SUBHU   Va, Vb, Rc`
10_2	01_2	`SUBHU.B Va, Vb, Rc`
10_2	10_2	`SUBHU.H Va, Vb, Rc`
11_2	00_2	`SUBHU   Va, Vb, Vc`
11_2	01_2	`SUBHU.B Va, Vb, Vc`
11_2	10_2	`SUBHU.H Va, Vb, Vc`
01_2	00_2	`SUBHU/F Va, Vb, Vc`
01_2	01_2	`SUBHU.B/F Va, Vb, Vc`
01_2	10_2	`SUBHU.H/F Va, Vb, Vc`

7.13.11 SUBHR - Signed half subtract with rounding

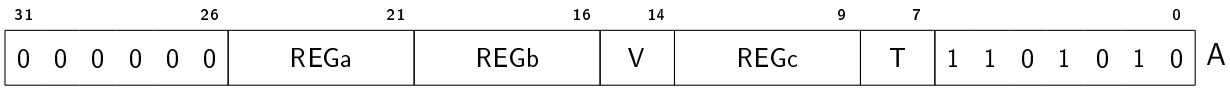
Requires: SM

Compute the rounded half difference of two signed integer operands.

Operation

`a` $\leftarrow$ `(int(b) - int(c) + 1)` $>>_s$ `1`

Encoding



Variants

V	T	Assembler
00_2	00_2	`SUBHR   Ra, Rb, Rc`
00_2	01_2	`SUBHR.B Ra, Rb, Rc`
00_2	10_2	`SUBHR.H Ra, Rb, Rc`
10_2	00_2	`SUBHR   Va, Vb, Rc`
10_2	01_2	`SUBHR.B Va, Vb, Rc`
10_2	10_2	`SUBHR.H Va, Vb, Rc`
11_2	00_2	`SUBHR   Va, Vb, Vc`
11_2	01_2	`SUBHR.B Va, Vb, Vc`
11_2	10_2	`SUBHR.H Va, Vb, Vc`
01_2	00_2	`SUBHR/F Va, Vb, Vc`
01_2	01_2	`SUBHR.B/F Va, Vb, Vc`
01_2	10_2	`SUBHR.H/F Va, Vb, Vc`

7.13.12 SUBHUR - Unsigned half subtract with rounding

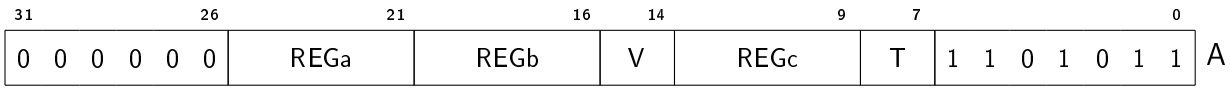
Requires: SM

Compute the rounded half difference of two unsigned integer operands.

Operation

```
a ← (uint(b) - uint(c) + 1) >> 1
```

Encoding



Variants

V	T	Assembler
00_2	00_2	`SUBHUR  Ra, Rb, Rc`
00_2	01_2	`SUBHUR.B Ra, Rb, Rc`
00_2	10_2	`SUBHUR.H Ra, Rb, Rc`
10_2	00_2	`SUBHUR  Va, Vb, Rc`
10_2	01_2	`SUBHUR.B Va, Vb, Rc`
10_2	10_2	`SUBHUR.H Va, Vb, Rc`
11_2	00_2	`SUBHUR  Va, Vb, Vc`
11_2	01_2	`SUBHUR.B Va, Vb, Vc`
11_2	10_2	`SUBHUR.H Va, Vb, Vc`
01_2	00_2	`SUBHUR/F Va, Vb, Vc`
01_2	01_2	`SUBHUR.B/F Va, Vb, Vc`
01_2	10_2	`SUBHUR.H/F Va, Vb, Vc`

7.13.13 MULQ - Multiply Q-numbers

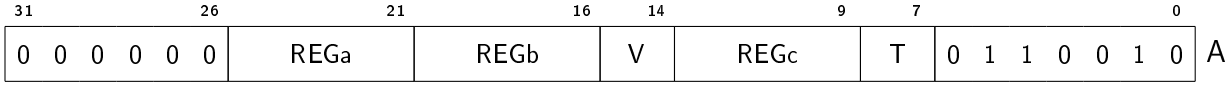
Requires: SM

Compute the product of two fixed point operands, with saturation.

Operation

```
prod ← int(b) × int(c)
a ← sat(prod >>_s (bits-1), bits)
```

Encoding



Variants

V	T	Assembler
00_2	00_2	`MULQ Ra, Rb, Rc`
00_2	01_2	`MULQ.B Ra, Rb, Rc`
00_2	10_2	`MULQ.H Ra, Rb, Rc`
10_2	00_2	`MULQ Va, Vb, Rc`
10_2	01_2	`MULQ.B Va, Vb, Rc`
10_2	10_2	`MULQ.H Va, Vb, Rc`
11_2	00_2	`MULQ Va, Vb, Vc`
11_2	01_2	`MULQ.B Va, Vb, Vc`
11_2	10_2	`MULQ.H Va, Vb, Vc`
01_2	00_2	`MULQ/F Va, Vb, Vc`
01_2	01_2	`MULQ.B/F Va, Vb, Vc`
01_2	10_2	`MULQ.H/F Va, Vb, Vc`

7.13.14 MULQR - Multiply Q-numbers with rounding

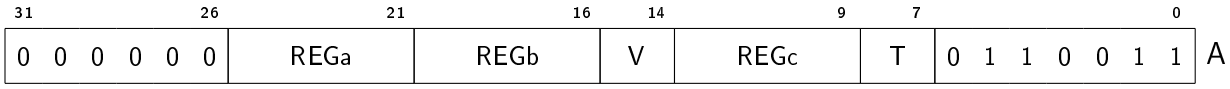
Requires: SM

Compute the rounded product of two fixed point operands, with saturation.

Operation

```
prod ← int(b) × int(c) + 1<<(bits-2)
a ← sat(prod >>s (bits-1), bits)
```

Encoding



Variants

V	T	Assembler
00_2	00_2	`MULQR  Ra, Rb, Rc`
00_2	01_2	`MULQR.B Ra, Rb, Rc`
00_2	10_2	`MULQR.H Ra, Rb, Rc`
10_2	00_2	`MULQR  Va, Vb, Rc`
10_2	01_2	`MULQR.B Va, Vb, Rc`
10_2	10_2	`MULQR.H Va, Vb, Rc`
11_2	00_2	`MULQR  Va, Vb, Vc`
11_2	01_2	`MULQR.B Va, Vb, Vc`
11_2	10_2	`MULQR.H Va, Vb, Vc`
01_2	00_2	`MULQR/F Va, Vb, Vc`
01_2	01_2	`MULQR.B/F Va, Vb, Vc`
01_2	10_2	`MULQR.H/F Va, Vb, Vc`

7.14 Processor control and status

7.14.1 XCHGSR - Exchange system register

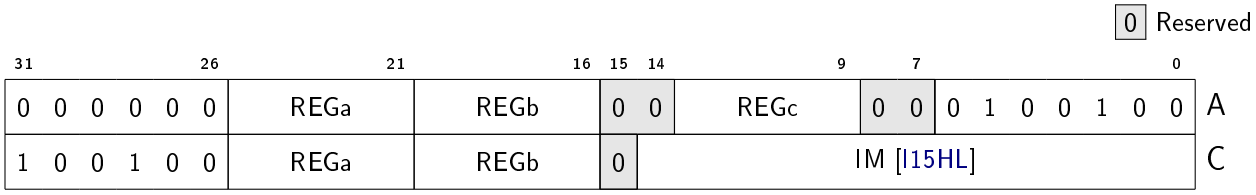
First move the value of the system register given by the second source operand to the destination operand, unless the destination operand is the Z register.

Then move the value of the first source operand to the system register given by the second source operand, unless the first source operand is the Z register.

Operation

```
if REGa ≠ 00000₂ then
  a ← SR[c]
if REGb ≠ 00000₂ then
  SR[c] ← b
```

Encoding



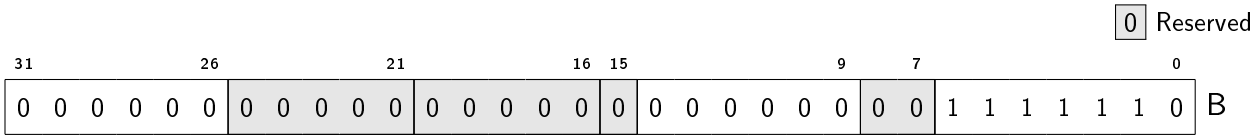
Variants

Fmt	Assembler
A	`XCHGSR Ra, Rb, Rc`
C	`XCHGSR Ra, Rb, #imm`

7.14.2 WAIT - Enter standby mode

Request that the pipeline is paused and placed in standby mode. The operation is implementation dependent, and may involve entering a low power mode. The pipeline is restarted when an external event, such as an interrupt, occurrs.

Encoding



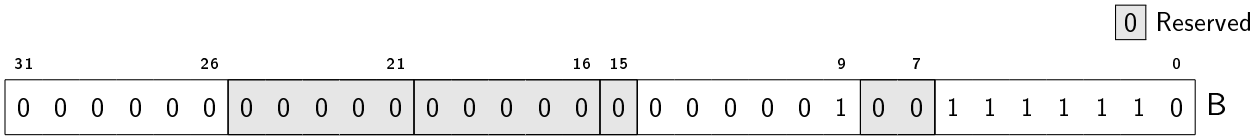
Variants

Assembler
WAIT

7.14.3 SYNC - Synchronize

Ensure that all instructions preceding the SYNC instruction appear to have completed before the SYNC instruction completes, and that no subsequent instructions are initiated by the processor until after the SYNC instruction completes. When the SYNC instruction completes, all external accesses caused by instructions preceding the SYNC instruction will have been performed with respect to all other mechanisms that access memory.

Encoding



Variants

Assembler
SYNC

> **Note**
>
> The instruction may take considerable (but finite) time to complete.

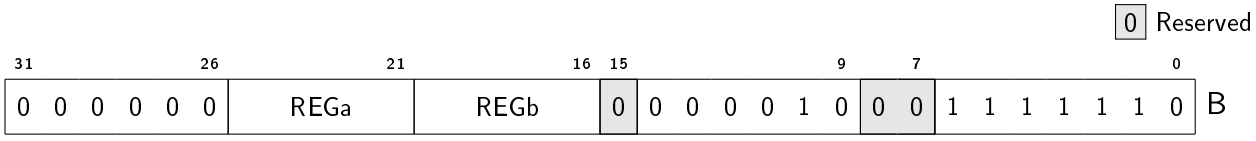
7.14.4 CCTRL - Cache control

Perform cache control. The cache control operation is specified by source operand B. If the cache control operation requires an additional operand (e.g. a memory address) it is passed in operand A. The instruction always generates a result that is written to the destination operand.

Operation

```
if b = 00000000₁₆ then
  Invalidate entire instruction cache
else if b = 00000001₁₆ then
  Invalidate entire data cache
else if b = 00000002₁₆ then
  Invalidate entire branch predictor
else if b = 00000100₁₆ then
  Invalidate instruction cache line containing address a
else if b = 00000101₁₆ then
  Invalidate data cache line containing address a
else if b = 00000201₁₆ then
  Flush entire data cache
else if b = 00000301₁₆ then
  Flush data cache line containing address a
a ← a
```

Encoding



Variants

Assembler
CCTRL Ra, Rb

> **Note**
>
> For many cache control operations it is suitable to pass register Z as operand A.

Chapter 8

System registers

This chapter describes all the system registers of the MRISC32 instruction set.

> **TODO**
>
> *Describe how to access the registers, and the rules for different kinds of registers (e.g. ordering).*

8.1 Identification

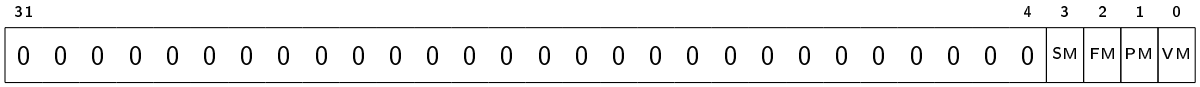
8.1.1 CPU_FEATURES_0

Number	R	W	Name
0000_{16}	✓		CPU feature flags register 0

Description

When a flag is set in this register, it indicates that the corresponding functionality is implemented.

Fields



VM (bit 0) Vector operation module implementation bit.

0: The Vector operation module is not implemented.

1: The Vector operation module is implemented.

PM (bit 1) Packed operation module implementation bit.

0: The Packed operation module is not implemented.

1: The Packed operation module is implemented.

FM (bit 2) Floating-point module implementation bit.

0: The Floating-point module is not implemented.

1: The Floating-point module is implemented.

SM (bit 3) Saturating and halving arithmetic module implementation bit.

0: The Saturating and halving arithmetic module is not implemented.

1: The Saturating and halving arithmetic module is implemented.

8.1.2 MAX_VL

Number	R	W	Name
0010_{16}	✓		Maximum vector length

Description

The maximum vector length for vector operations.

Fields

MAX_VL (bits <31:0>) Maximum vector length (number of elements in each vector register).

For implementations that advertise support for the Vector operation module (VM), this value shall be a power of two, and at least 16.

For implementations that do not support vector operations, this value shall be zero (0).

Chapter 9

Conventions

9.1 Instruction aliases

This section defines valid assembler aliases for common operations that are implemented using more generic instructions.

It is recommended that instruction aliases are used in place of their generic counterparts in most situations, such as assembler code generated by a compiler or a disassembler.

The main purposes of the instruction aliases are to improve the readability of assembler programs and listings, and to make it easier to write assembler programs.

9.1.1 ASR - Arithmetic shift right

Shift signed integer to the right.

Syntax:

```
asr  ra, rb, #shift  ; Immediate
asr  ra, rb, rc      ; Register
```

Expands to:

```
ebf  ra, rb, #<shift:0>  ; Immediate
ebf  ra, rb, rc          ; Register
```

9.1.2 B - Branch

Unconditionally branch to a PC-relative target.

Syntax:

```
b  #target
```

Expands to:

```
j  pc, #target@pc
```

> **Note**
>
> The branch range for the B alias is PC +/-4MiB.
>
> This alias is preferred to alternatives based on conditional branch instructions with known conditions (such as using BZ with the Z register as the condition).

9.1.3 BL - Branch and link

Unconditionally branch and link to a PC-relative target.

Syntax:

```
bl  #target
```

Expands to:

```
jl  pc, #target@pc
```

> **Note**
>
> The branch range for the BL alias is PC +/-4MiB.

9.1.4 CALL - Call a subroutine

Call a subroutine, with full 32-bit address range.

Syntax:

```
call  #target@pc  ; PC-relative
call  #target     ; Absolute
```

Expands to:

```
; PC-relative
addpchi  lr, #target@pchi
jl       lr, #target+4@pclo

; Absolute
ldi      lr, #target@hi
jl       lr, #target@lo
```

9.1.5 GETSR - Get system register

Move the value of a system register to a general purpose register.

Syntax:

```
getsr  ra, #sr_reg_no  ; Immediate
getsr  ra, rc          ; Register
```

Expands to:

```
xchgsr  ra, z, #sr_reg_no  ; Immediate
xchgsr  ra, z, rc          ; Register
```

9.1.6 LSL - Logic shift left

Shift integer to the left.

Syntax:

```
lsl  ra, rb, #shift  ; Immediate
lsl  ra, rb, rc      ; Register
```

Expands to:

```
mkbf  ra, rb, #<shift:0>  ; Immediate
mkbf  ra, rb, rc          ; Register
```

9.1.7 LSR - Logic shift right

Shift unsigned integer to the right.

Syntax:

```
lsr  ra, rb, #shift  ; Immediate
lsr  ra, rb, rc      ; Register
```

Expands to:

```
ebfu  ra, rb, #<shift:0>  ; Immediate
ebfu  ra, rb, rc          ; Register
```

9.1.8 MOV - Move

Move value to register.

Syntax:

```
mov  ra, #value  ; Immediate
mov  ra, rb      ; Register
```

Expands to:

```
or  ra, (v)z, #value  ; Immediate
or  ra, (v)z, rb      ; Register
```

> **Note**
>
> The immediate form of the MOV alias is mostly useful for vector target registers. For scalar targer registers the LDI instruction is more suitable since it has a wider immediate range.

9.1.9 NOP - No operation

Perform no operation.

Syntax:

```
nop
```

Expands to:

```
or  z, z, z
```

9.1.10 RET - Return

Retrun from a subroutine (jump to the address pointed to by LR).

Syntax:

```
ret
```

Expands to:

```
j  lr, #0
```

9.1.11 SETSR - Set system register

Move the value of a general purpose register to a system register.

Syntax:

```
setsr  rb, #sr_reg_no  ; Immediate
setsr  rb, rc          ; Register
```

Expands to:

```
xchgsr  z, rb, #sr_reg_no   ; Immediate
xchgsr  z, rb, rc           ; Register
```

9.1.12 TAIL - Tail call

Make a tail call to a sibling routine, with full 32-bit address range.

Syntax:

```
tail  #target@pc   ; PC-relative
tail  #target      ; Absolute
```

Expands to:

```
; PC-relative
addpchi  r15, #target@pchi
j        r15, #target+4@pclo

; Absolute
ldi      r15, #target@hi
j        r15, #target@lo
```

> **Note**
>
> The TAIL alias implicitly clobbers the R15 register. When using the TAIL alias for making tail calls, this is well defined behavior since R15 is defined as an intra-procedure call scratch register in the recommended calling convention.

9.2 Canonical constructs

Certain operations can be done in several ways with (more or less) equivalent effect, but for the sake of hardware implementation efficiency this section defines the preferred way for those operations.

TBD

Chapter 10

Application Binary Interface

This chapter contains recommendations for platform application binary interfaces (ABIs). It is not a complete ABI specification.

10.1 Calling convention

10.1.1 Scalar registers

Register	Alias	Role and rule
Z	R0	Always zero (read only)
R1-R8		Function arguments / results
R9-R14		Temporary registers
R15		Intra-procedure call scratch register / temporary register
R16-R26		Callee-saved registers
TP	R27	Thread pointer (callee-saved)
FP	R28	Frame pointer (callee-saved)
SP	R29	Stack pointer (callee-saved)
LR	R30	Link register (callee-saved)
VL	R31	Vector length (callee-saved)

Function arguments / results

The first arguments to a function are passed in registers R1 to R8. How many registers are used depends on the number of arguments and the types of the arguments. For more information, see 10.1.4.

Likewise function results are returned in R1 to R8. For more information, see 10.1.5.

These registers may also be used as temporary registers.

Temporary registers

Temporary registers are not guaranteed to be preserved across function call boundaries, and thus need not be preserved by the callee.

Callee-saved registers

The contents of callee-saved registers must be preserved by a function. This is normally done by the function prologue and epilogue by storing and restoring the registers to and from the stack.

Intra-procedure call scratch register

The intra-procedure call scratch register may be used for call target address calculations. It may also be used as a temporary register.

Thread pointer

The thread pointer may be used by systems that need to provide fast access to thread local data. Otherwise it may be used as a general purpose register.

The thread pointer is a callee-saved register.

Frame pointer

TBD

Stack pointer

Upon function entry, the stack pointer contains the address of the top of the stack. For more information, see 10.1.3.

The stack pointer is a callee-saved register.

Link register

The link register contains the return address to the caller.

The link register is a callee-saved register.

Vector length

The vector length is a callee-saved register.

10.1.2 Vector registers

Register	Alias	Role and rule
VZ	V0	Always zero (read only)
V1-V8		Function arguments / results
V9-V31		Temporary registers

Function arguments / results

The first vector arguments to a function are passed in registers V1 to V8. How many registers are used depends on the number of arguments.

Likewise function vector results are returned in V1 to V8.

These registers may also be used as temporary registers.

Temporary registers

All vector registers are temporary registers, and thus need not be preserved by the callee.

10.1.3 Stack

TBD

10.1.4 Function arguments

TBD

10.1.5 Function results

TBD

10.2 Data organization

10.2.1 Endianness

Data fields are stored in memory using little endian representation. Thus the least significant byte of a data field is at the lowest byte address that the data field occupies in memory.

10.2.2 Alignment

Data fields that are one, two or four bytes in size shall be aligned to a memory address that is divisable by the data field size.

Data fields that are larger than four bytes in size shall be aligned to a memory address that is divisable by four.

Type	Size (bytes)	Alignment (bytes)
byte	1	1
half-word	2	2
word	4	4
double-word	8	4
quad-word	16	4

Appendix A

Alphabetical list of instructions

This non-normative section lists all the instructions of the MRISC32 ISA.

Legend:

- Base — Part of the Base architecture
- PM — Requires the Packed operation module
- FM — Requires the Floating-point module
- SM — Requires the Saturating and halving arithmetic module

Instruction	Base	PM	FM	SM	Name
ADD	✓				Add
ADDH				✓	Signed half add
ADDHR				✓	Signed half add with rounding
ADDHU				✓	Unsigned half add
ADDHUR				✓	Unsigned half add with rounding
ADDPC	✓				Add PC and immediate
ADDPCHI	✓				Add PC and high immediate
ADDS				✓	Signed add with saturation
ADDSU				✓	Unsigned add with saturation
AND	✓				Bitwise and
BGE	✓				Branch if greater than or equal
BGT	✓				Branch if greater than
BLE	✓				Branch if less than or equal
BLT	✓				Branch if less than
BNS	✓				Branch if not set
BNZ	✓				Branch if not zero
BS	✓				Branch if set

(continued)

Instruction	Base	PM	FM	SM	Name
BZ	✓				Branch if zero
CCTRL	✓				Cache control
CLZ	✓				Count leading zeros
CRC32	✓				Calculate CRC-32 checksum
CRC32C	✓				Calculate CRC-32C checksum
DIV	✓				Signed divide
DIVU	✓				Unsigned divide
EBF	✓				Extract bit field
EBFU	✓				Extract bit field unsigned
FADD			✓		Floating-point add
FDIV			✓		Floating-point divide
FMAX			✓		Floating-point maximum
FMIN			✓		Floating-point minimum
FMUL			✓		Floating-point multiply
FPACK		✓	✓		Floating-point pack
FSEQ			✓		Floating-point set if equal
FSLE			✓		Floating-point set if less than or equal
FSLT			✓		Floating-point set if less than
FSNE			✓		Floating-point set if not equal
FSORD			✓		Floating-point set if ordered
FSUB			✓		Floating-point subtract
FSUNORD			✓		Floating-point set if unordered
FTOI			✓		Floating-point to signed integer
FTOIR			✓		Floating-point to signed integer with rounding
FTOU			✓		Floating-point to unsigned integer
FTOUR			✓		Floating-point to unsigned integer with rounding
FUNPH		✓	✓		Floating-point unpack high
FUNPL		✓	✓		Floating-point unpack low
IBF	✓				Insert bit field
ITOF			✓		Signed integer to floating-point
J	✓				Jump
JL	✓				Jump and link
LDB	✓				Load signed byte
LDEA	✓				Load effective address
LDH	✓				Load signed half-word
LDI	✓				Load immediate
LDUB	✓				Load unsigned byte
LDUH	✓				Load unsigned half-word

(continued)

Instruction	Base	PM	FM	SM	Name
LDW	✓				Load word
LDWPC	✓				Load word PC-relative
MADD	✓				Multiply and add
MAX	✓				Signed maxiumum
MAXU	✓				Unsigned maximum
MIN	✓				Signed minimum
MINU	✓				Unsigned minimum
MKBF	✓				Make bit field
MUL	✓				Multiply
MULHI	✓				Signed multiply high
MULHIU	✓				Unsigned multiply high
MULQ				✓	Multiply Q-numbers
MULQR				✓	Multiply Q-numbers with rounding
OR	✓				Bitwise or
PACK		✓			Pack
PACKHI		✓			Pack high
PACKHIR		✓		✓	Signed pack high with rounding
PACKHIUR		✓		✓	Unsigned pack high with rounding
PACKS		✓		✓	Signed pack with saturation
PACKSU		✓		✓	Unsigned pack with saturation
POPCNT	✓				Population count
REM	✓				Signed remainder
REMU	✓				Unsigned remainder
REV	✓				Reverse bits
SEL	✓				Bitwise select
SEQ	✓				Set if equal
SHUF	✓				Shuffle bytes
SLE	✓				Set if less than or equal
SLEU	✓				Set if less than or equal unsigned
SLT	✓				Set if less than
SLTU	✓				Set if less than unsigned
SNE	✓				Set if not equal
STB	✓				Store byte
STH	✓				Store half-word
STW	✓				Store word
STWPC	✓				Store word PC-relative
SUB	✓				Subtract
SUBH				✓	Signed half subtract
SUBHR				✓	Signed half subtract with rounding
SUBHU				✓	Unsigned half subtract

(continued)

Instruction	Base	PM	FM	SM	Name
SUBHUR				✓	Unsigned half subtract with rounding
SUBS				✓	Signed subtract with saturation
SUBSU				✓	Unsigned subtract with saturation
SYNC	✓				Synchronize
UTOF			✓		Unsigned integer to floating-point
WAIT	✓				Enter standby mode
XCHGSR	✓				Exchange system register
XOR	✓				Bitwise exclusive or

Appendix B

Opcode list

This non-normative section lists all the opcodes, used and vacant, of the MRISC32 ISA.

Legend:

- **OP** — The instruction operation identifier
- **FN** — The instruction function identifier (extended operation)
- Base — Part of the Base architecture
- PM — Requires the Packed operation module
- FM — Requires the Floating-point module
- SM — Requires the Saturating and halving arithmetic module

B.1 Format A opcodes

OP	Instruction	Base	PM	FM	SM	Name
4	STB	✓				Store byte
5	STH	✓				Store half-word
6	STW	✓				Store word
7	-					
8	LDB	✓				Load signed byte
9	LDH	✓				Load signed half-word
10	LDW	✓				Load word
11	-					
12	LDUB	✓				Load unsigned byte
13	LDUH	✓				Load unsigned half-word
14	-					

(continued)

OP	Instruction	Base	PM	FM	SM	Name
15	LDEA	✓				Load effective address
16	AND	✓				Bitwise and
17	OR	✓				Bitwise or
18	XOR	✓				Bitwise exclusive or
19	EBF	✓				Extract bit field
20	EBFU	✓				Extract bit field unsigned
21	MKBF	✓				Make bit field
22	ADD	✓				Add
23	SUB	✓				Subtract
24	MIN	✓				Signed minimum
25	MAX	✓				Signed maxiumum
26	MINU	✓				Unsigned minimum
27	MAXU	✓				Unsigned maximum
28	SEQ	✓				Set if equal
29	SNE	✓				Set if not equal
30	SLT	✓				Set if less than
31	SLTU	✓				Set if less than unsigned
32	SLE	✓				Set if less than or equal
33	SLEU	✓				Set if less than or equal unsigned
34	SHUF	✓				Shuffle bytes
35	-					
36	XCHGSR	✓				Exchange system register
37	-					
38	-					
39	MUL	✓				Multiply
40	DIV	✓				Signed divide
41	DIVU	✓				Unsigned divide
42	REM	✓				Signed remainder
43	REMU	✓				Unsigned remainder
44	MADD	✓				Multiply and add
45	-					
46	SEL	✓				Bitwise select
47	IBF	✓				Insert bit field
48	MULHI	✓				Signed multiply high
49	MULHIU	✓				Unsigned multiply high
50	MULQ				✓	Multiply Q-numbers
51	MULQR				✓	Multiply Q-numbers with rounding
52	-					
53	-					
54	-					

(continued)

OP	Instruction	Base	PM	FM	SM	Name
55	-					
56	-					
57	-					
58	PACK		✓			Pack
59	PACKS		✓		✓	Signed pack with saturation
60	PACKSU		✓		✓	Unsigned pack with saturation
61	PACKHI		✓			Pack high
62	PACKHIR		✓		✓	Signed pack high with rounding
63	PACKHIUR		✓		✓	Unsigned pack high with rounding
64	FMIN			✓		Floating-point minimum
65	FMAX			✓		Floating-point maximum
66	FSEQ			✓		Floating-point set if equal
67	FSNE			✓		Floating-point set if not equal
68	FSLT			✓		Floating-point set if less than
69	FSLE			✓		Floating-point set if less than or equal
70	FSUNORD			✓		Floating-point set if unordered
71	FSORD			✓		Floating-point set if ordered
72	ITOF			✓		Signed integer to floating-point
73	UTOF			✓		Unsigned integer to floating-point
74	FTOI			✓		Floating-point to signed integer
75	FTOU			✓		Floating-point to unsigned integer
76	FTOIR			✓		Floating-point to signed integer with rounding
77	FTOUR			✓		Floating-point to unsigned integer with rounding
78	FPACK		✓	✓		Floating-point pack
79	-					
80	FADD			✓		Floating-point add
81	FSUB			✓		Floating-point subtract
82	FMUL			✓		Floating-point multiply
83	FDIV			✓		Floating-point divide
84	-					
85	-					
86	-					
87	-					
88	-					
89	-					
90	-					
91	-					
92	-					

(continued)

OP	Instruction	Base	PM	FM	SM	Name
93	-					
94	-					
95	-					
96	ADDS				✓	Signed add with saturation
97	ADDSU				✓	Unsigned add with saturation
98	ADDH				✓	Signed half add
99	ADDHU				✓	Unsigned half add
100	ADDHR				✓	Signed half add with rounding
101	ADDHUR				✓	Unsigned half add with rounding
102	SUBS				✓	Signed subtract with saturation
103	SUBSU				✓	Unsigned subtract with saturation
104	SUBH				✓	Signed half subtract
105	SUBHU				✓	Unsigned half subtract
106	SUBHR				✓	Signed half subtract with rounding
107	SUBHUR				✓	Unsigned half subtract with rounding
108	-					
109	-					
110	-					
111	-					
112	-					
113	-					
114	-					
115	-					
116	-					
117	-					
118	-					
119	-					
120	-					
121	-					
122	-					
123	-					

B.2 Format B opcodes

OP	FN	Instruction	Base	PM	FM	SM	Name
124	0	REV	✓				Reverse bits
124	1	CLZ	✓				Count leading zeros

(continued)

OP	FN	Instruction	Base	PM	FM	SM	Name
124	2	POPCNT	✓				Population count
124	3	-					
124	4	-					
124	5	-					
124	6	-					
124	7	-					
124	8	-					
124	9	-					
124	10	-					
124	11	-					
124	12	-					
124	13	-					
124	14	-					
124	15	-					
124	16	-					
124	17	-					
124	18	-					
124	19	-					
124	20	-					
124	21	-					
124	22	-					
124	23	-					
124	24	-					
124	25	-					
124	26	-					
124	27	-					
124	28	-					
124	29	-					
124	30	-					
124	31	-					
124	32	-					
124	33	-					
124	34	-					
124	35	-					
124	36	-					
124	37	-					
124	38	-					
124	39	-					
124	40	-					
124	41	-					

(continued)

OP	FN	Instruction	Base	PM	FM	SM	Name
124	42	-					
124	43	-					
124	44	-					
124	45	-					
124	46	-					
124	47	-					
124	48	-					
124	49	-					
124	50	-					
124	51	-					
124	52	-					
124	53	-					
124	54	-					
124	55	-					
124	56	-					
124	57	-					
124	58	-					
124	59	-					
124	60	-					
124	61	-					
124	62	-					
124	63	-					
125	0	FUNPL		✓	✓		Floating-point unpack low
125	1	FUNPH		✓	✓		Floating-point unpack high
125	2	-					
125	3	-					
125	4	-					
125	5	-					
125	6	-					
125	7	-					
125	8	-					
125	9	-					
125	10	-					
125	11	-					
125	12	-					
125	13	-					
125	14	-					
125	15	-					
125	16	-					
125	17	-					

(continued)

OP	FN	Instruction	Base	PM	FM	SM	Name
125	18	-					
125	19	-					
125	20	-					
125	21	-					
125	22	-					
125	23	-					
125	24	-					
125	25	-					
125	26	-					
125	27	-					
125	28	-					
125	29	-					
125	30	-					
125	31	-					
125	32	-					
125	33	-					
125	34	-					
125	35	-					
125	36	-					
125	37	-					
125	38	-					
125	39	-					
125	40	-					
125	41	-					
125	42	-					
125	43	-					
125	44	-					
125	45	-					
125	46	-					
125	47	-					
125	48	-					
125	49	-					
125	50	-					
125	51	-					
125	52	-					
125	53	-					
125	54	-					
125	55	-					
125	56	-					
125	57	-					

(continued)

OP	FN	Instruction	Base	PM	FM	SM	Name
125	58	-					
125	59	-					
125	60	-					
125	61	-					
125	62	-					
125	63	-					
126	0	WAIT	✓				Enter standby mode
126	1	SYNC	✓				Synchronize
126	2	CCTRL	✓				Cache control
126	3	-					
126	4	-					
126	5	-					
126	6	-					
126	7	-					
126	8	CRC32C	✓				Calculate CRC-32C checksum
126	9	CRC32	✓				Calculate CRC-32 checksum
126	10	-					
126	11	-					
126	12	-					
126	13	-					
126	14	-					
126	15	-					
126	16	-					
126	17	-					
126	18	-					
126	19	-					
126	20	-					
126	21	-					
126	22	-					
126	23	-					
126	24	-					
126	25	-					
126	26	-					
126	27	-					
126	28	-					
126	29	-					
126	30	-					
126	31	-					
126	32	-					
126	33	-					

(continued)

OP	FN	Instruction	Base	PM	FM	SM	Name
126	34	-					
126	35	-					
126	36	-					
126	37	-					
126	38	-					
126	39	-					
126	40	-					
126	41	-					
126	42	-					
126	43	-					
126	44	-					
126	45	-					
126	46	-					
126	47	-					
126	48	-					
126	49	-					
126	50	-					
126	51	-					
126	52	-					
126	53	-					
126	54	-					
126	55	-					
126	56	-					
126	57	-					
126	58	-					
126	59	-					
126	60	-					
126	61	-					
126	62	-					
126	63	-					
127	0	-					
127	1	-					
127	2	-					
127	3	-					
127	4	-					
127	5	-					
127	6	-					
127	7	-					
127	8	-					
127	9	-					

(continued)

OP	FN	Instruction	Base	PM	FM	SM	Name
127	10	-					
127	11	-					
127	12	-					
127	13	-					
127	14	-					
127	15	-					
127	16	-					
127	17	-					
127	18	-					
127	19	-					
127	20	-					
127	21	-					
127	22	-					
127	23	-					
127	24	-					
127	25	-					
127	26	-					
127	27	-					
127	28	-					
127	29	-					
127	30	-					
127	31	-					
127	32	-					
127	33	-					
127	34	-					
127	35	-					
127	36	-					
127	37	-					
127	38	-					
127	39	-					
127	40	-					
127	41	-					
127	42	-					
127	43	-					
127	44	-					
127	45	-					
127	46	-					
127	47	-					
127	48	-					
127	49	-					

(continued)

OP	FN	Instruction	Base	PM	FM	SM	Name
127	50	-					
127	51	-					
127	52	-					
127	53	-					
127	54	-					
127	55	-					
127	56	-					
127	57	-					
127	58	-					
127	59	-					
127	60	-					
127	61	-					
127	62	-					
127	63	-					

B.3 Format C opcodes

OP	Instruction	Base	PM	FM	SM	Name
4	STB	✓				Store byte
5	STH	✓				Store half-word
6	STW	✓				Store word
7	-					
8	LDB	✓				Load signed byte
9	LDH	✓				Load signed half-word
10	LDW	✓				Load word
11	-					
12	LDUB	✓				Load unsigned byte
13	LDUH	✓				Load unsigned half-word
14	-					
15	LDEA	✓				Load effective address
16	AND	✓				Bitwise and
17	OR	✓				Bitwise or
18	XOR	✓				Bitwise exclusive or
19	EBF	✓				Extract bit field
20	EBFU	✓				Extract bit field unsigned
21	MKBF	✓				Make bit field
22	ADD	✓				Add

(continued)

OP	Instruction	Base	PM	FM	SM	Name
23	SUB	✓				Subtract
24	MIN	✓				Signed minimum
25	MAX	✓				Signed maxiumum
26	MINU	✓				Unsigned minimum
27	MAXU	✓				Unsigned maximum
28	SEQ	✓				Set if equal
29	SNE	✓				Set if not equal
30	SLT	✓				Set if less than
31	SLTU	✓				Set if less than unsigned
32	SLE	✓				Set if less than or equal
33	SLEU	✓				Set if less than or equal unsigned
34	SHUF	✓				Shuffle bytes
35	-					
36	XCHGSR	✓				Exchange system register
37	-					
38	-					
39	MUL	✓				Multiply
40	DIV	✓				Signed divide
41	DIVU	✓				Unsigned divide
42	REM	✓				Signed remainder
43	REMU	✓				Unsigned remainder
44	MADD	✓				Multiply and add
45	-					
46	SEL	✓				Bitwise select
47	IBF	✓				Insert bit field

B.4 Format D opcodes

OP	Instruction	Base	PM	FM	SM	Name
0	J	✓				Jump
1	JL	✓				Jump and link
2	LDWPC	✓				Load word PC-relative
3	STWPC	✓				Store word PC-relative
4	ADDPC	✓				Add PC and immediate
5	ADDPCHI	✓				Add PC and high immediate
6	LDI	✓				Load immediate

B.5 Format E opcodes

OP	Instruction	Base	PM	FM	SM	Name
0	BZ	✓				Branch if zero
1	BNZ	✓				Branch if not zero
2	BS	✓				Branch if set
3	BNS	✓				Branch if not set
4	BLT	✓				Branch if less than
5	BGE	✓				Branch if greater than or equal
6	BLE	✓				Branch if less than or equal
7	BGT	✓				Branch if greater than

Appendix C

Alphabetical list of system registers

This non-normative section lists all the system registers of the MRISC32 ISA.

Register	Number	R	W	Name
CPU_FEATURES_0	0000_{16}	✓		CPU feature flags register 0
MAX_VL	0010_{16}	✓		Maximum vector length

Appendix D

Examples

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

D.1 Basic operations

D.1.1 Push/pop stack

```
non_leaf_function:
    ; Push registers r16, r17 and lr onto the stack
    add   sp, sp, #-12
    stw   r16, [sp, #0]
    stw   r17, [sp, #4]
    stw   lr, [sp, #8]

    ; ...

    ; Pop registers and return from function
    ldw   lr, [sp, #8]
    ldw   r17, [sp, #4]
    ldw   r16, [sp, #0]
    add   sp, sp, #12
    ret                     ; Alias for j lr, #0
```

D.1.2 Simple loop

```
    ldi   r1, #loop_count   ; r1 holds the loop counter
loop:
    ; ...
    add   r1, r1, #-1       ; Decrement the loop counter
    bnz   r1, loop          ; Branch if r1 != 0
```

D.1.3 Conditional selection

```
    sne   r4, r1, #42
    sel   r4, r2, r3        ; r4 = (r1 != 42) ? r2 : r3
```

D.2 Vector operation

D.2.1 saxpy

Several BLAS routines, including saxpy, are easily vectorized for the MRISC32 instruction set.

```
; 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:
;   r1 - n
;   r2 - a
;   r3 - x
;   r4 - y

saxpy:
    bz    r1, 2f          ; Nothing to do?
    getsr vl, #0x10       ; Query the maximum vector length
1:
    minu  vl, vl, r1      ; Define the operation vector length
    sub   r1, r1, vl      ; Decrement loop counter
    ldw   v1, [r3, #4]    ; Load x (element stride = 4 bytes)
    ldw   v2, [r4, #4]    ; Load y
    fmul  v1, v1, r2      ; x * a
    fadd  v1, v1, v2      ; + y
    stw   v1, [r4, #4]    ; Store y
    ldea  r3, [r3, vl*4]  ; Increment address (x)
    ldea  r4, [r4, vl*4]  ; Increment address (y)
    bnz   r1, 1b
2:
    ret
```

D.2.2 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:
;   r1 - n
;   r2 - t0
;   r3 - dt
;   r4 - x
;   r5 - y

lerp:
    bz    r1, 2f            ; Nothing to do?

    getsr vl, #0x10         ; Query the maximum vector length

    add   r6, r4, #4        ; r6 = &x[1]
    itof  r7, vl, z

    ldea  v1, [z, #1]       ; v1 = [0, 1, 2, ...]
    itof  v1, v1, z
    fmul  v1, v1, r3        ; v1 = dt * [0.0, 1.0, 2.0, ...]

    fmul  r7, r3, r7        ; r7 = dt * maximum vector length
1:
    minu  vl, vl, r1        ; Define the operation vector length
    sub   r1, r1, 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, [r4, v2*4]    ; Load x[k]
    ldw   v5, [r6, 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, [r5, #4]      ; Store y (element stride = 4 bytes)

    ldea  r5, [r5, vl*4]    ; Increment address (y)
    fadd  v1, v1, r7        ; Increment t
    bnz   r1, 1b
2:
    ret
```

D.2.3 Reverse bytes

Reversing a byte array (e.g. for horizontal mirroring of an image) can be achieved by copying 32-bit words in reverse order (using a negative stride when storing the words), in combination with reversing the bytes of each individual word using the SHUF instruction.

```
; void revbytes(size_t n, const uint8_t *x, uint8_t *y)
; {
;   for (size_t i = 0; i < n; i++)
;     y[n-1-i] = x[i];
; }
;
; Register arguments:
;   r1 - n
;   r2 - x
;   r3 - y
;
; Assumptions:
;   n is a multiple of 4

revbytes:
    bz    r1, 2f           ; Nothing to do?
    add   r4, r1, #-4
    add   r3, r3, r4       ; r3 = &y[n-4]
    lsr   r1, r1, #2       ; r1 = number of words
    getsr vl, #0x10        ; Query the maximum vector length
    lsl   r4, vl, #2       ; r4 = 4 * max vector length
1:
    minu  vl, vl, r1       ; Define the operation vector length
    sub   r1, r1, vl       ; Decrement loop counter
    ldw   v1, [r2, #4]     ; Load x (element stride = 4 bytes)
    shuf  v1, v1, #0b000001010011  ; Reverse bytes of each word
    stw   v1, [r3, #-4]    ; Store y (element stride = -4 bytes)
    add   r2, r2, r4       ; Increment address (x)
    sub   r3, r3, r4       ; Decrement address (y)
    bnz   r1, 1b
2:
    ret
```

Bibliography

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