SuperH

SuperH™ (SH) 64-Bit RISC Series

SH-5 CPU Core, Volume 2: SHmedia

Last updated 22 February 2002

Issued by the SuperH Documentation Group on behalf of SuperH, Inc.

SuperH, Inc.
San Jose, U.S.A. - Bristol, United Kingdom - Tokyo, Japan

www.superh.com

SuperH

Contents

Preface

This document is part of the SuperH SH-5 CPU core documentation suite detailed below. Comments on this or other books in the documentation suite should be made by contacting your local sales office or distributor.

SuperH SH-5 document identification and control

Each book in the documentation suite carries a unique identifier in the form:

05-CC-nnnnn Vx.x

Where, *n* is the document number and *x.x* is the revision.

Whenever making comments on a SuperH SH-5 document the complete identification 05-CC-1000n Vx.x should be quoted.

SuperH SH-5 CPU core documentation suite

The SuperH SH-5 CPU core documentation suite comprises the following volumes:

- SH-5 CPU Core, Volume 1: Architecture (05-CC-10001)
- SH-5 CPU Core, Volume 2: SHmedia (05-CC-10002)
- SH-5 CPU Core, Volume 3: SHcompact (05-CC-10003)
- SH-5 CPU Core, Volume 4: Implementation (05-CC-10004)

SuperH

SHmedia specification

1

1.1 Overview

The behavior of instructions is specified using a simple notational language to describe the effects of each instruction on the architectural state of the machine.

The language consists of the following features:

- A simple variable and type system.
- Expressions.
- Statements.
- Notation for the architectural state of the machine.
- An abstract sequential model of instruction execution.

These features are described in the following sections. Additional mechanisms are defined to model memory, control registers, configuration registers, synchronization instructions, cache instructions and floating-point. The final section gives example instruction specifications.

Each instruction is described using informal text as well as the formal notational language. Sometimes it is inappropriate for one of these descriptions to convey the full semantics. In such cases these two descriptions must be taken together to constitute the full specification. In the case of an ambiguity or a conflict, the notational language takes precedence over the text.

1.2 Variables and types

Variables are used to hold state. The type of a variable determines the set of values that the variable can take and the available operators to manipulate that variable.

The scalar types are integer, boolean and bit-field. The integer type is the only arithmetic type provided and obeys standard mathematical properties. Booleans are used to represent conditions that can be either true or false. Bit-fields are used to define a bit-accurate representation of a value. Although integers and bit-fields are distinct types, bit-fields can be read as integer values and written with integer values using the simple mappings defined in *Section 1.2.3: Bit-fields on page 3*.

The architectural state of the machine is represented by a set of variables. Each of these variables has an associated type, which is either a bit-field or an array of bit-fields. Additional variables are used to hold temporary values. The type of temporary variables is implicit, and determined by context rather than explicit declaration. The type of a temporary variable is an integer, a boolean or an array of these.

1.2.1 Integer

An integer variable can take the value of any mathematical integer. No limits are imposed on the range of integers supported. Integers obey their standard mathematical properties. Integer operations do not overflow. The integer operators are defined so that singularities do not occur. For example, no definition is given to the result of divide by zero; the operator is simply not available when the divisor is zero.

The representation of literal integer values is achieved using the following notations:

- Decimal numbers are represented by the regular expression: {0-9}+
- Hexadecimal numbers are represented by the regular expression: 0x{0-9a-fA-F}+
- Binary numbers are represented by the regular expression: 0b{0-1}+

These notations are standard and map onto integer values in the obvious way. Underscore characters ('_') can be inserted into any of the above literal representations. These do not change the represented value but can be used as spacers to aid readability.

The notations allow only zero and positive numbers to be represented directly. A monadic integer negation operator can subsequently be used to derive a negative value.

1.2.2 Boolean

A boolean variable can take two values:

- Boolean false. The literal representation of boolean false is 'FALSE'.
- Boolean true. The literal representation of boolean true is 'TRUE'.

1.2.3 Bit-fields

Bit-fields are provided to define 'bit-accurate' storage.

Bit-fields containing arbitrary numbers of bits are supported. A bit-field of b bits contains bits numbered from 0 (the least significant bit) up to b-1 (the most significant bit). Each bit can take the value 0 or the value 1. Bit-fields are mapped to, and from, integers in the usual way. If bit i of a b-bit, bit-field, where i is in [0, b), is set then it contributes 2^i to the integral value of the bit-field. The integral value of the bit-field as a whole is an integer in the range [0, 2^b).

When a bit-field is read, it gives its integral value. When a bit-field is written with an integral value, the integer must be in the range of values supported by the bit-field. Typically, the only operations applied directly to bit-fields are conversions to other types.

1.2.4 Arrays

One-dimensional arrays of the above types are also available. Indexing into an n-element array A is achieved using the notation A[i] where A is an array of some type and i is an integer in the range [0, n). This selects the $i^{th.}$ element of the array A. If i is zero this selects the first entry, and if i is n-1 then this selects the last entry. The type of the selected element is the base type of the array.

Multi-dimensional arrays are not provided.

1.2.5 Floating point values

Floating-point types and operators are not provided. Instead, the value in a floating-point register is represented as a bit-field. The organization of the bit-field is consistent with the IEEE754 format.

When a floating-point register is read, an integral representation of that bit-pattern is returned. When an integral value is written into a floating-point register, the value written is the bit-pattern of that integer. Thus, reading and writing is achieved as bit-pattern transfers, and not by interpreting the bit-patterns as real numbers.

The language does not provide direct means to interpret these bit-patterns as real numbers. Instead, functions are provided which give the required functionality. For example, arithmetic on real numbers is represented using a function notation.

1.3 Expressions

Expressions are constructed from monadic operators, dyadic operators and functions applied to variable and literal values.

There are no defined precedence and associativity rules for the operators. Parentheses are used to specify the expression unambiguously.

Sub-expressions can be evaluated in any order. If a particular evaluation order is required, then sub-expressions must be split into separate statements.

1.3.1 Integer arithmetic operators

Since the notation uses straightforward mathematical integers, the set of standard mathematical operators is available and already defined.

The standard dyadic operators are listed in *Table 1*.

Operation	Description
i + j	Integer addition
i - j	Integer subtraction
i × j	Integer multiplication
i / j	Integer division
i \ j	Integer remainder

Table 1: Standard dyadic operators

The standard monadic operators are described in *Table 2*.

Operator	Description
- i	Integer negation
|i|	Integer modulus

Table 2: Standard monadic Operators

The division operator truncates towards zero. The remainder operator is consistent with this. The sign of the result of the remainder operator follows the sign of the dividend. Division or remainder with a divisor of zero results in a singularity, and its behavior is not defined.

For a numerator (n) and a denominator (d), the following properties hold where $d \neq 0$:

$$n = d \times (n/d) + (n\backslash d)$$
$$(-n)/d = -(n/d) = n/(-d)$$
$$(-n)\backslash d = -(n\backslash d)$$
$$n\backslash(-d) = n\backslash d$$
$$0 \leq (n\backslash d) < d \quad \text{where} \quad n \geq 0 \quad \text{and} \quad d > 0$$

1.3.2 Integer shift operators

The available integer shift operators are listed in *Table 3*.

Operation	Description
n << b	Integer left shift
n >> b	Integer right shift

Table 3: Shift operators

The shift operators are defined on integers as follows where $b \geq 0$:

$$n \ll b = n \times 2^b$$

$$n \gg b = \begin{cases} n/2^b & \text{where} \quad n \geq 0 \\ (n - 2^b + 1)/2^b & \text{where} \quad n < 0 \end{cases}$$

Note that right shifting rounds the result towards minus infinity. This contrasts with division, which rounds towards zero, and is the reason why the right shift definition is separate for positive and negative n.

1.3.3 Integer bitwise operators

The available integer bitwise operators are listed in *Table 4*.

Operation	Description
$i \wedge j$	Integer bitwise AND
$i \vee j$	Integer bitwise OR
$i \oplus j$	Integer bitwise XOR
$\sim i$	Integer bitwise NOT
$n_{<b\ FOR\ m>}$	Integer field extraction: extract m bits starting at bit b from integer n
$n_{<b>}$	Integer field extraction: extract 1 bit starting at bit b from integer n

Table 4: Bitwise operators

In order to define bitwise operations all integers are considered as having an infinitely long two's complement representation. Bit 0 is the least significant bit of this representation, bit 1 is the next higher bit, and so on. The value of bit b, for all b such that $b \geq 0$, in integer n is given by:

$$BIT(n, b) = (n / 2^{b}) \backslash 2 \qquad \text{where} \qquad n \geq 0$$

$$BIT(n, b) = 1 - BIT(-(n + 1), b) \qquad \text{where} \qquad n < 0$$

Care must be taken whenever the infinitely long two's complement representation of a negative number is constructed. This representation will contain an infinite number of higher bits with the value 1 representing the sign. Typically, a subsequent conversion operation is used to discard these upper bits and return the result back to a finite value.

Bitwise AND ($\wedge$), OR ($\vee$), XOR ($\oplus$) and NOT ($\sim$) are defined on integers as follows, where b takes all values such that $b \geq 0$:

$$BIT(i \wedge j, b) = BIT(i, b) \times BIT(j, b)$$

$$BIT(i \vee j, b) = BIT(i \wedge j, b) + BIT(i \oplus j, b)$$

$$BIT(i \oplus j, b) = (BIT(i, b) + BIT(j, b)) \backslash 2$$

$$BIT(\sim i, b) = 1 - BIT(i, b)$$

Note that bitwise NOT of any finite positive i will result in a value containing an infinite number of higher bits with the value 1 representing the sign.

Bitwise extraction is defined on integers as follows, where $b \geq 0$ and $m > 0$:

$$n_{\langle b \text{ FOR } m \rangle} = (n \gg b) \wedge (2^m - 1)$$

$$n_{\langle b \rangle} = n_{\langle b \text{ FOR } 1 \rangle}$$

The result of $n_{<b \text{ FOR } m>}$ is an integer in the range $[0, 2^m)$.

1.3.4 Relational operators

Relational operators are defined to compare integral values and give a boolean result.

Operation	Description
i = j	Result is true if i is equal to j, otherwise false
i ≠ j	Result is true if i is not equal to j, otherwise false
i < j	Result is true if i is less than j, otherwise false
i > j	Result is true if i is greater than j, otherwise false
i ≤ j	Result is true if i is less than or equal to j, otherwise false
i ≥ j	Result is true if i is greater than or equal to j, otherwise false

Table 5: Relational operators

1.3.5 Boolean operators

Boolean operators are defined to perform logical AND, OR, XOR and NOT. These operators have boolean sources and result. Additionally, the conversion operator INT is defined to convert a boolean source into an integer result.

Operation	Description
i AND j	Result is true if i and j are both true, otherwise false
i OR j	Result is true if either/both i and j are true, otherwise false

Table 6: Boolean operators

Operation	Description
i XOR j	Result is true if exactly one of i and j are true, otherwise false
NOT i	Result is true if i is false, otherwise false
INT i	Result is 0 if i is false, otherwise 1

Table 6: Boolean operators

1.3.6 Single-value functions

In some cases it is inconvenient or inappropriate to describe an expression directly in the specification language. In these cases a function call is used to reference the undescribed behavior.

A single-value function evaluates to a single value (the result), which can be used in an expression. The type of the result value can be determined by the expression context from which the function is called. There are also multiple-value functions which evaluate to multiple values. These are only available in an assignment context, and are described in *Section 1.4.2: Assignment on page 14*.

Functions can contain side-effects.

Scalar conversions

Two monadic functions are defined to support conversion from finite-precision signed and unsigned number ranges. For a finite-precision integer representation containing n bits, the signed number range is $[-2^{n-1}, 2^{n-1})$ while the unsigned number range is $[0, 2^{n})$.

These functions are often used to convert between bit-fields and integer values.

Function	Description
$\text{ZeroExtend}_n(i)$	Convert integer i to an n-bit 2's complement unsigned range
$\text{SignExtend}_n(i)$	Convert integer i to an n-bit 2's complement signed range

Table 7: Integer range conversion operators

These two functions are defined as follows, where n > 0:

$$\text{ZeroExtend}_n(i) = i_{\langle 0 \text{ FOR } n \rangle}$$

$$\text{SignExtend}_n(i) = \begin{cases} i_{\langle 0 \text{ FOR } n \rangle} & \text{where} \quad i_{\langle n-1 \rangle} = 0 \\ i_{\langle 0 \text{ FOR } (n-1) \rangle} - 2^n & \text{where} \quad i_{\langle n-1 \rangle} = 1 \end{cases}$$

For syntactic convenience, conversion functions are also defined for converting an integer to a single bit and to a 64-bit register. *Table 8* shows the additional functions provided.

Operation	Description
Bit(i)	Convert lowest bit of integer i to a 1-bit unsigned value This is a convenient notation for $i_{<0>}$
Register(i)	Convert lowest 64 bits of integer i to a 64-bit unsigned value This is a convenient notation for $i_{<0 \text{ FOR } 64>}$

Table 8: Bit and register conversion operators

Multimedia conversions

Conversion functions are defined to aid the handling of multimedia types. Multimedia types are held in a packed form within 64-bit registers. The supported formats are:

- 8 x 8-bit unsigned values, 8 x 8-bit signed values.
- 4 x 16-bit unsigned values, 4 x 16-bit signed values.
- 2 x 32-bit unsigned values, 2 x 32-bit signed values.

Conversions are available to convert from packed integer representations in these formats to arrays of integer values, and vice versa. The integer array has the same number of elements as there are values in the multimedia format. For a multimedia format constructed from n-bit values (where n is 8, 16 or 32), the array will have 64/n elements.

The following conversions are provided from packed integer representations to an array of integer values:

Operation	Description
$\text{MultiZeroExtend}_n(A)$	Interpret integer A as a 64-bit packed representation, and return an array of 64/n integers where each element has an n-bit unsigned integer value (n is 8, 16 or 32)
$\text{MultiSignExtend}_n(A)$	Interpret integer A as a 64-bit packed representation, and return an array of 64/n integers where each element has an n-bit signed integer value (n is 8, 16 or 32)

Table 9: Conversion to multimedia types

The conversions are defined as follows, where i takes all values in [0, 64/n):

$$(\text{MultiZeroExtend}_n(A))[i] = \text{ZeroExtend}_n(A \gg (i \times n))$$

$$(\text{MultiSignExtend}_n(A))[i] = \text{SignExtend}_n(A \gg (i \times n))$$

The following conversion is defined from an array of integer values to a packed integer representation:

Operation	Description
$\text{MultiRegister}_n(a)$	Convert the lowest n bits of each element in the array a of 64/n integers to a 64-bit packed representation and return as an integer value (n is 8, 16 or 32)

Table 10: Conversion from multimedia types

The conversion is defined as follows, where i takes all values in [0, 64/n):

$$(\text{MultiRegister}_n(a))_{\langle (i \times n) \text{ FOR } n \rangle} = \text{ZeroExtend}_n(a[i])$$

$$(\text{MultiRegister}_n(a)) \gg 64 = 0$$

The effect of the second clause in the MultiRegister definition is to define that the returned integer value is in an unsigned 64-bit range. This ensures that the integer can be directly assigned to a 64-bit, bit-field.

Saturation

Two monadic functions are defined to support saturation of integers within representations of finite-precision signed and unsigned number spaces:

Function	Description
$\text{UnsignedSaturate}_n(i)$	Saturate integer i to an n-bit 2's complement unsigned range
$\text{SignedSaturate}_n(i)$	Saturate integer i to an n-bit 2's complement signed range

Table 11: Integer saturation operators

These two functions are defined as follows, where $n > 0$:

$$\text{UnsignedSaturate}_n(i) = \begin{cases} 0 & \text{where} \quad i < 0 \\ i & \text{where} \quad 0 \le i < 2^n \\ 2^n - 1 & \text{where} \quad 2^n \le i \end{cases}$$

$$\text{SignedSaturate}_n(i) = \begin{cases} -2^{n-1} & \text{where} \quad i < -2^{n-1} \\ i & \text{where} \quad -2^{n-1} \le i < 2^{n-1} \\ 2^{n-1} - 1 & \text{where} \quad 2^{n-1} \le i \end{cases}$$

Packed byte extraction

Two monadic functions are defined for convenient manipulation of packed byte data:

Function	Description
$\text{LowerBytes}_n(i)$	Returns just the lower n (out of 8) bytes of i
$\text{UpperBytes}_n(i)$	Returns just the upper n (out of 8) bytes of i (without shifting)

Table 12: Integer saturation operators

These two functions are defined as follows, where n is in [0, 8]:

$$\text{LowerBytes}_n(i) = i \wedge (2^{n \times 8} - 2^0)$$

$$\text{UpperBytes}_n(i) = i \wedge (2^{64} - 2^{64-(n \times 8)})$$

Floating-point conversions

The specification language manipulates floating-point values as integers containing the associated IEEE754 bit-pattern. The layout of these bit-patterns is described in *Volume 1, Chapter 3: Data representation*. The language does not support a floating-point type.

Conversion functions are defined to support floating-point. Floating-point values are held as either scalar values in a single register, or vector values in multiple registers. The available register formats are:

- One 32-bit value in a single-precision register.
- One 64-bit value in a double-precision register.
- Two 32-bit values in a pair of single-precision registers.
- Four 32-bit values in a four-entry vector of single-precision registers.
- Sixteen 32-bit values in a four-by-four matrix of single-precision registers.

These register formats are mapped onto the same floating-point register file. This mapping is described in *Volume 1, Chapter 2: Architectural state*.

Conversions are available to convert between register bit-fields in these formats and integers or arrays of integers holding the appropriate IEEE754 bit-patterns.

The following conversions are provided to convert from floating-point registers:

Operation	Description
$FloatValue_{32}(r)$	Convert a single-precision floating-point register into a 32-bit integer bit-pattern.
$FloatValue_{64}(r)$	Convert a double-precision floating-point register into a 64-bit integer bit-pattern.
$FloatValuePair_{32}(r)$	Convert a pair of single-precision floating-point registers into an array of 2 x 32-bit integer bit-patterns.
$FloatValueVector_{32}(r)$	Convert a 4-entry vector of single-precision floating-point registers into an array of 4 x 32-bit integer bit-patterns.
$FloatValueMatrix_{32}(r)$	Convert a 16-entry matrix of single-precision floating-point registers into an array of 16 x 32-bit integer bit-patterns.

Table 13: Conversion from floating-point register formats

The following conversions are provided to convert to floating-point registers:

Operation	Description
$FloatRegister_{32}(i)$	Convert a 32-bit integer bit-pattern into a single-precision floating-point register.
$FloatRegister_{64}(i)$	Convert a 64-bit integer bit-pattern into a double-precision floating-point register.
$FloatRegisterPair_{32}(a)$	Convert an array of 2 x 32-bit integer bit-patterns into a pair of single-precision floating-point registers.
$FloatRegisterVector_{32}(a)$	Convert an array of 4 x 32-bit integer bit-patterns into a 4-entry vector of single-precision floating-point registers.
$FloatRegisterMatrix_{32}(a)$	Convert an array of 16 x 32-bit integer bit-patterns into a 16-entry matrix of single-precision floating-point registers.

Table 14: Conversion to floating-point register formats

1.4 Statements

An instruction specification consists of a sequence of statements. These statements are processed sequentially in order to specify the effect of the instruction on the architectural state of the machine. The available statements are discussed in this section.

Each statement has a semi-colon terminator. A sequence of statements can be aggregated into a statement block using '{' to introduce the block and '}' to terminate the block. A statement block can be used anywhere that a statement can.

1.4.1 Undefined behavior

The statement:

```
UNDEFINED();
```

indicates that the resultant behavior is architecturally undefined.

A particular implementation can choose to specify an implementation-defined behavior in such cases. It is very likely that any implementation-defined behavior will vary from implementation to implementation. Exploitation of

implementation-defined behavior should be avoided to allow software to be portable between implementations.

In cases where architecturally undefined behavior can occur in user mode, the implementation will ensure that implemented behavior does not break the protection model. Thus, the implemented behavior will be some execution flow that is permitted for that user mode thread.

1.4.2 Assignment

The '←' operator is used to denote assignment of an expression to a variable. An example assignment statement is:

```
variable ← expression;
```

The expression can be constructed from variables, literals, operators and functions as described in *Section 1.3: Expressions on page 4*. The expression is fully evaluated before the assignment takes place. The variable can be an integer, a boolean, a bit-field or an array of one of these types.

Assignment to architectural state

This is where the variable is part of the architectural state (as described in *Table 16: Scalar architectural state on page 18*). The type of the expression and the type of the variable must match.

Assignment to a temporary

Alternatively, if the variable is not part of the architectural state, then it is a temporary variable. The type of the variable is determined by the type of expression. A temporary variable must be assigned to, before it is used in the instruction specification.

Assignment of an undefined value

An assignment of the following form results in a variable being initialized with an architecturally undefined value:

```
variable ← UNDEFINED;
```

After assignment the variable will hold a value which is valid for its type. However, the value is architecturally undefined. The actual value can be unpredictable; that is to say the value indicated by UNDEFINED can vary with each use of UNDEFINED. Architecturally-undefined values can occur in both user and privileged modes.

A particular implementation can choose to specify an implementation-defined value in such cases. It is very likely that any implementation-defined values will vary from implementation to implementation. Exploitation of implementation-defined values should be avoided to allow software to be portable between implementations.

Assignment of multiple values

Multi-value functions are used to return multiple values, and are only available when used in a multiple assignment context. The syntax consists of a list of comma-separated variables, an assignment symbol followed by a function call. The function is evaluated and returns multiple results into the variables listed. The number of variables and the number of results of the function must match. The assigned variables must all be distinct (that is, no aliases).

For example, a two-valued assignment from a function call with 3 parameters can be represented as:

```
variable1, variable2 ← call(param1, param2, param3);
```

1.4.3 Conditional

Conditional behavior is specified using 'IF', 'ELSE IF' and 'ELSE'.

Conditions are expressions that result in a boolean value. If the condition after an 'IF' is true, then its block of statements is executed and the whole conditional then completes. If the condition is false, then any 'ELSE IF' clauses are processed, in turn, in the same fashion. If no conditions are met and there is an 'ELSE' clause then its block of statements is executed. Finally, if no conditions are met and there is no 'ELSE' clause, then the statement has no effect apart from the evaluation of the condition expressions.

The 'ELSE IF' and 'ELSE' clauses are optional. In ambiguous cases, the 'ELSE' matches with the nearest 'IF'.

For example:

```
IF (condition1)
  block1
ELSE IF (condition2)
  block2
ELSE
  block3
```

1.4.4 Repetition

Repetitive behavior is specified using the following construct:

```
REPEAT i FROM m FOR n STEP s
  block
```

The block of statements is iterated n times, with the integer i taking the values:

m, m + s, m + 2s, m + 3s, up to m + (n - 1) × s.

The behavior is equivalent to textually writing the block n times with i being substituted with the appropriate value in each copy of the block.

The value of n must be greater or equal to 0. The values of the expressions for m, n and s must be constant across the iteration. The integer i must not be assigned to within the iterated block. The 'STEP s' can be omitted in which case the step-size takes the default value of 1.

1.4.5 Exceptions

Exception handling is triggered by a **THROW** statement. When an exception is thrown, no further statements are executed from the instruction specification and control passes to an exception handler. The actions associated with the launch of the handler are not shown in the instruction specification, but are described separately in *Volume 1, Chapter 16: Event handling*.

There are two forms of throw statement:

```
THROW type;
```

and:

```
THROW type, value;
```

where **type** indicates the type of exception which is launched, and **value** is an optional argument to the exception handling sequence.

The set of exceptions used in the instruction specification are shown in *Table 15*.

Exception name	Cause	Optional argument
BREAK	Break	Not required
EXECPROT	Execute without permission	Faulty instruction address

Table 15: Exception list

Exception name	Cause	Optional argument
FPUDIS	FPU is disabled	Not required
FPUEXC	FPU exception	FPSCR value
IADDERR	Fetch from a malformed or misaligned address	Faulty instruction address
ITLBMISS	Fetch with no MMU mapping	Faulty instruction address
RESINST	Reserved instruction or execution of a privileged instruction in user mode	Not required
RADDERR	Read from a malformed or misaligned address	Faulty data address
RTLBMISS	Read with no MMU mapping	Faulty data address
READPROT	Read without permission	Faulty data address
TRAP	Trap	Trap constant value
WADDERR	Write to a malformed or misaligned address	Faulty data address
WTLBMISS	Write with no MMU mapping	Faulty data address
WRITEPROT	Write without permission	Faulty data address

Table 15: Exception list

The full set of exceptions is described in *Volume 1, Chapter 16: Event handling*.

1.4.6 Procedures

Procedure statements contain a procedure name followed by a list of comma-separated arguments contained within parentheses followed by a semi-colon. The execution of procedures typically causes side-effects to the architectural state of the machine.

Procedures are generally used where it is difficult or inappropriate to specify the effect of an instruction using the abstract execution model. A fuller description of the effect of the instruction will be given in the surrounding text.

An example procedure with two parameters is:

```
proc(param1, param2);
```

1.5 Architectural state

The architectural state is described in *Volume 1, Chapter 2: Architectural state*. The notations used in the model to refer to this state are summarized in *Table 16* and *Table 17*. Each item of scalar architectural state is a bit-field of a particular width. Each item of array architectural state is an array of bit-fields of a particular width.

Architectural state	Type is a bit-field containing:	Description
MD (SR.MD)	1 bit	User (0) or privileged (1) mode
ISA	1 bit	SHcompact (0) or SHmedia (1) instruction set
PC	64 bits	64-bit program counter
FPSCR	32 bits	32-bit floating-point status and control register
R_i where i is in [0, 63]	64 bits	64 x 64-bit general purpose registers R_{63} reads as zero; writes to R_{63} are ignored
TR_i where i is in [0, 7]	64 bits	8 x 64-bit target address registers
FR_i where i is in [0, 63] DR_{2i} where i is in [0, 31]	32 bits 64 bits	64 x 32-bit floating-point registers 32 x 64-bit floating-point registers
CR_i where i is in [0, 63]	64 bits	64 x 64-bit control registers

Table 16: Scalar architectural state

Architectural state	Type is an array of bit-fields each containing:	Description
FP_{2i} where i is in [0, 31]	32 bits	32 pairs of 32-bit floating-point registers
FV_{4i} where i is in [0, 15]	32 bits	16 vectors of 4 x 32-bit floating-point registers
$MTRX_{16i}$ where i is in [0, 3]	32 bits	4 matrices of 16 x 32-bit floating-point registers

Table 17: Array architectural state

Architectural state	Type is an array of bit-fields each containing:	Description
MEM[i] where i is in [0, 2^{64})	8 bits	2^{64} bytes of memory
CFG[i] where i is in [0, 2^{64})	64 bits	2^{64} x 64-bit configuration registers

Table 17: Array architectural state

FR, FP, FV, MTRX and DR provide different views of the same architectural state.

There is no implicit meaning to the value held by the collection of bits in a register. The interpretation of the register is supplied by each instruction that reads or writes the register value.

PC denotes the program counter of the currently executing instruction. PC' denotes the program counter of the next instruction that is to be executed.

Implemented control registers are also given specific names, and these are listed in *Volume 1, Chapter 9: SHmedia system instructions*.

1.6 Memory model

Instruction specification uses a simple model of memory. It assumes, for example, that any caches have no architectural visibility. For typical well-disciplined instruction sequences these effects will not be architecturally visible. However, a fuller description of the behavior in other cases is defined by the text of the architecture manual.

MEM is an array of bytes indexed by an effective address. Elements in arrays are selected using array indexing notation: MEM[i] selects the $i^{th.}$ entry in the MEM array. The total range of array indices into MEM is [0, 2^{64}), though not all of this memory is available on all implementations.

Array slicing can be used to view an array as consisting of elements of a larger size. The notation MEM[s FOR n], where n > 0, denotes a memory slice containing the elements MEM[s], MEM[s+1] through to MEM[s+n-1]. The type of this slice is a bit-field exactly large enough to contain a concatenation of the n selected elements. In this case it contain 8n bits since the base type of MEM is byte.

The order of the concatenation depends on the endianness of the processor:

- If the processor is operating in a little endian mode, the concatenation order obeys the following condition as i (the byte number) varies in the range [0, n):

$$(\text{MEM}[s \text{ FOR } n])_{\langle 8i \text{ FOR } 8\rangle} = \text{MEM}[s + i]$$

  This equivalence states that byte number i, using little endian byte numbering (that is, byte 0 is bits 0 to 7), in the bit-field MEM[s FOR n] is the $i^{th.}$ byte in memory counting upwards from MEM[s].

- If the processor is operating in a big endian mode, the concatenation order obeys the following condition as i (the byte number) varies in the range [0, n):

$$(\text{MEM}[s \text{ FOR } n])_{\langle 8(n-1-i) \text{ FOR } 8\rangle} = \text{MEM}[s + i]$$

  This equivalence states that byte number i, using big endian byte numbering (that is, byte 0 is bits 8n-8 to 8n-1), in the bit-field MEM[s FOR n] is the $i^{th.}$ byte in memory counting upwards from MEM[s].

For syntactic convenience, functions and procedures are provided to read, write and swap memory. The basic primitives support aligned accesses. Misaligned read and write primitives support the instructions for misaligned load and store.

Additionally, mechanisms are provided for reading and writing pairs of values. Pair access requires that each half of the pair is endianness converted separately, and that the lower half is written into memory at the provided address while the upper half is written into that address plus the object size. This maintains the ordering of the halves of the pair as they are transferred between registers and memory. Pair access is used only for loading and storing pairs of single-precision floating-point registers (see *Volume 1, Chapter 8: SHmedia floating-point*).

1.6.1 Support functions

The specification of the memory instructions relies on the support functions listed in *Table 18*. These functions are used to model the behavior of the memory management unit described in *Volume 1, Chapter 17: Memory management*.

Function	Description
MalformedAddress(address)	Returns true if the provided address is a malformed address (that is, outside of the implemented part of the effective address space).
MMU()	Returns true if the MMU is enabled.
DataAccessMiss(address)	Returns true if the provided address does not have a mapping for a data access.
InstFetchMiss(address)	Returns true if the provided address does not have a mapping for an instruction fetch.
InstInvalidateMiss(address)	Returns true if the provided address does not have a mapping for an instruction invalidation.
InstPrefetchMiss(address)	Returns true if the provided address does not have a mapping for an instruction prefetch.
ReadProhibited(address)	Returns true if the provided address has no read permission for the current privilege.
WriteProhibited(address)	Returns true if the provided address has no write permission for the current privilege.
ExecuteProhibited(address)	Returns true if the provided address has no execute permission for the current privilege.
IsLittleEndian()	Returns true if processor is little endian.

Table 18: Support functions for memory access

More detailed properties of translation miss detection are not modelled here. The conditions that determine whether an access is a translation miss or a hit depend on the MMU and cache. There is considerable flexibility in the organization of the MMU and cache, and this typically involves properties specific to the implementation. Software is often constructed so that translation miss handling occurs transparently with respect to program execution.

DataAccessMiss is used to check for the absence of a data translation. This function is used for all data accesses when the MMU is enabled. Three different functions are

used to check for the absence of an instruction translation. InstFetchMiss is used for instruction fetches, InstInvalidateMiss for instruction invalidations and InstPrefetchMiss for instruction prefetches. These cases differ in exception handling:

1 Instruction fetch causes a translation miss when executing an instruction without a translation.

2 Instruction invalidation (ICBI) causes translation misses on some, but not all, implementations, depending on the MMU and cache organization (see *Volume 1, Chapter 17: Memory management* and *Volume 1, Chapter 18: Caches*).

3 Instruction prefetch (PREFI) silently drops any translation misses.

1.6.2 Reading memory

Functions are provided to read memory.

Function	Description
ReadMemory_n(address)	Aligned memory read of an n-bit value
ReadMemoryPair_n(address)	Aligned memory read of a pair of n-bit values
ReadMemoryLow_n(address)	Misaligned memory read of an n-bit value (address is low byte)
ReadMemoryHigh_n(address)	Misaligned memory read of an n-bit value (address is high byte)

Table 19: Support functions to read memory

The ReadMemory_n function takes an integer parameter to indicate the address being accessed. The number of bits being read (n) is one of 8, 16, 32 or 64 bits. The required bytes are read from memory, interpreted according to endianness, and an integer result returns the read bit-field value. If the read memory value is to be interpreted as signed, then a sign-extension should be used on the result. The assignment:

```
result ← ReadMemory_n(a);
```

is equivalent to:

```
width ← n >> 3;
IF (MalformedAddress(a) OR ((a∧(width-1)) ≠ 0)) THROW RADDERR,a;
IF (MMU() AND DataAccessMiss(a)) THROW RTLBMISS,a;
IF (MMU() AND ReadProhibited(a)) THROW READPROT,a;
result ← MEM[a FOR width];
```

ReadMemoryPair$_n$ reads a pair of n-bit values from memory, and returns the pair as an array of two integers. The alignment check requires alignment for a 2n-bit access. The access maintains the ordering of the two halves of the pair, with endianness applied separately to each half. The assignment:

```
result ← ReadMemoryPair_n(a);
```

is equivalent to:

```
width ← n >> 3;
pairwidth ← width << 1;
IF (MalformedAddress(a) OR ((a∧(pairwidth-1)) ≠ 0))
  THROW RADDERR,a;
IF (MMU() AND DataAccessMiss(a)) THROW RTLBMISS,a;
IF (MMU() AND ReadProhibited(a)) THROW READPROT,a;
result[0] ← MEM[a FOR width];
result[1] ← MEM[a+width FOR width];
```

ReadMemoryLow$_n$ and ReadMemoryHigh$_n$ support misaligned access. In this case, the width can be any whole number of bytes in the range [1, 8] and there is no alignment check. The address parameter to ReadMemoryLow$_n$ is the address of the lowest byte to be read. The assignment:

```
result ← ReadMemoryLow_n(a);
```

is equivalent to:

```
width ← n >> 3;
IF (MalformedAddress(a)) THROW RADDERR,a;
IF (MMU() AND DataAccessMiss(a)) THROW RTLBMISS,a;
IF (MMU() AND ReadProhibited(a)) THROW READPROT,a;
result ← MEM[a FOR width];
```

The address parameter to ReadMemoryHigh$_n$ is the address of the highest byte to be read. The assignment:

```
result ← ReadMemoryHigh_n(a);
```

is equivalent to:

```
width ← n >> 3;
start ← (a - width) + 1;
IF (MalformedAddress(a)) THROW RADDERR,a;
IF (MMU() AND DataAccessMiss(a)) THROW RTLBMISS,a;
IF (MMU() AND ReadProhibited(a)) THROW READPROT,a;
result ← MEM[start FOR width];
```

1.6.3 Prefetching memory

A function is provided to denote memory prefetch.

Function	Description
PrefetchMemory(address)	Memory prefetch

Table 20: Support procedure to prefetch memory

This is used for a software-directed data prefetch from a specified effective address. This is a hint to give advance notice that particular data will be required. It is implementation-specific as to whether a prefetch will be performed.

The statement:

```
result ← PrefetchMemory(a);
```

is equivalent to:

```
IF (NOT MalformedAddress(address))
  IF (NOT (MMU() AND DataAccessMiss(address)))
    IF (NOT (MMU() AND ReadProhibited(address)))
      PREFO(address);
result ← 0;
```

where PREFO is a cache operation defined in *Section 1.8: Cache model on page 28*. This function does not raise exceptions. PrefetchMemory evaluates to zero for syntactic convenience.

1.6.4 Writing memory

Procedures are provided to write memory.

Function	Description
$WriteMemory_n$(address, value)	Aligned memory write to an n-bit value
$WriteMemoryPair_n$(address, value)	Aligned memory write to a pair of n-bit values
$WriteMemoryLow_n$(address, value)	Misaligned memory write to an n-bit value (address is low byte)
$WriteMemoryHigh_n$(address, value)	Misaligned memory write to an n-bit value (address is high byte)

Table 21: Support procedures to write memory

The WriteMemory$_n$ procedure takes an integer parameter to indicate the address being accessed, followed by an integer parameter containing the value to be written. The number of bits being written (n) is one of 8, 16, 32 or 64 bits. The written value is interpreted as a bit-field of the required size; all higher bits of the value are discarded. The bytes are written to memory, ordered according to endianness. The statement:

```
WriteMemory_n(a, value);
```

is equivalent to:

```
width ← n >> 3;
IF (MalformedAddress(a) OR ((a∧(width-1)) ≠0)) THROW WADDERR,a;
IF (MMU() AND DataAccessMiss(a)) THROW WTLBMISS,a;
IF (MMU() AND WriteProhibited(a)) THROW WRITEPROT,a;
MEM[a FOR width] ← value<0 FOR n>;
```

WriteMemoryPair$_n$ writes an array of two integers to memory as a pair of n-bit values. The alignment check requires alignment for a 2n-bit access. The access maintains the ordering of the two halves of the pair, with endianness applied separately to each half. The statement:

```
WriteMemoryPair_n(a, value);
```

is equivalent to:

```
width ← n >> 3;
pairwidth ← width << 1;
IF (MalformedAddress(a) OR ((a∧(pairwidth-1)) ≠0))
  THROW WADDERR,a;
IF (MMU() AND DataAccessMiss(a)) THROW WTLBMISS,a;
IF (MMU() AND WriteProhibited(a)) THROW WRITEPROT,a;
MEM[a FOR width] ← (value[0])<0 FOR n>;
MEM[a+width FOR width] ← (value[1])<0 FOR n>;
```

WriteMemoryLow$_n$ and WriteMemoryHigh$_n$ support misaligned access. In this case, the width can be any whole number of bytes in the range [1, 8] and there is no alignment check. The address parameter to WriteMemoryLow$_n$ is the address of the lowest byte to be written. The statement:

```
WriteMemoryLow_n(a, value);
```

is equivalent to:

```
width ← n >> 3;
```

```
IF (MalformedAddress(a)) THROW WADDERR,a;
IF (MMU() AND DataAccessMiss(a)) THROW WTLBMISS,a;
IF (MMU() AND WriteProhibited(a)) THROW WRITEPROT,a;
MEM[a FOR width] ← value<0 FOR n>;
```

The address parameter to WriteMemoryHigh$_n$ is the address of the highest byte to be written. The statement:

```
WriteMemoryHigh_n(a, value);
```

is equivalent to:

```
width ← n >> 3;
start ← (a - width) + 1;
IF (MalformedAddress(a)) THROW WADDERR,a;
IF (MMU() AND DataAccessMiss(a)) THROW WTLBMISS,a;
IF (MMU() AND WriteProhibited(a)) THROW WRITEPROT,a;
MEM[start FOR width] ← value<0 FOR n>;
```

1.6.5 Swapping memory

A function is provided to swap values with memory locations.

Function	Description
SwapMemory$_n$(address, value)	Aligned memory swap

Table 22: Support function to swap memory

The SwapMemory$_n$ function takes an integer parameter to indicate the address being accessed, followed by an integer parameter containing the value to be written The number of bits being swapped (n) is one of 8, 16, 32 or 64 bits (although not all of these swap widths are provided by the architecture). The required bytes are read from memory, interpreted according to endianness, and an integer result returns the read bit-field value. The written value is interpreted as a bit-field of the required size; all higher bits of the value are discarded. The bytes are written to memory, ordered according to endianness.The read and write on memory are performed atomically with respect to other memory users.

The assignment:

```
result ← SwapMemory_n(a, value);
```

is equivalent to:

```
width ← n >> 3;
IF (MalformedAddress(a) OR ((a∧(width-1)) ≠0)) THROW WADDERR,a;
IF (MMU() AND DataAccessMiss(a)) THROW WTLBMISS,a;
IF (MMU() AND ReadProhibited(a)) THROW READPROT,a;
IF (MMU() AND WriteProhibited(a)) THROW WRITEPROT,a;
result ← MEM[a FOR width];
MEM[a FOR width] ← value<0 FOR n>;
```

If the read memory value is to be interpreted as a signed value, then a sign-extending conversion should be used on the result. There are no misaligned nor pair variants.

1.7 Sleep and synchronization operations

The SLEEP operation is used to enter sleep mode. The SYNCI and SYNCO operations are used to synchronize the instruction stream and operand data accesses, respectively. The effects of these operations are beyond the scope of the specification language, and are therefore modelled using procedure calls. The behavior of these procedure calls is elaborated in the text of the manual.

Procedure	Description
SLEEP()	Procedure to enter sleep mode
SYNCI()	Procedure to synchronize the instruction stream.
SYNCO()	Procedure to synchronize operand data.

Table 23: Procedures to model sleep and synchronization operations

1.8 Cache model

Cache operations are used to allocate, prefetch and cohere lines in caches. The effects of these operations are beyond the scope of the specification language, and are therefore modelled using procedure calls. The behavior of these procedure calls is elaborated in the text of the manual.

Procedure	Description
ALLOCO(address)	Procedure to allocate an operand cache block
ICBI(address)	Procedure to invalidate an instruction cache block.
OCBI(address)	Procedure to invalidate an operand cache block.
OCBP(address)	Procedure to purge an operand cache block.
OCBWB(address)	Procedure to write-back an operand cache block.
PREFI(address)	Procedure to prefetch an instruction cache block.
PREFO(address)	Procedure to prefetch an operand cache block.

Table 24: Procedures to model cache operations

1.9 Control register model

The control register file is denoted CR. A function called ReadControlRegister is provided to read control registers. The assignment:

```
result ← ReadControlRegister(index);
```

is equivalent to:

$\texttt{result} \leftarrow \texttt{CR}_{\texttt{index}}\texttt{;}$

A procedure called WriteControlRegister is provided to write control registers. The statement:

```
WriteControlRegister(index, value);
```

is equivalent to:

$\texttt{CR}_{\texttt{index}} \leftarrow \texttt{value;}$

Functions are used in the instruction specifications to determine undefined and privileged control registers, see *Table 25*.

Function	Description
IsUndefinedControlRegister(index)	Returns true if the index corresponds to an undefined control register.
IsPrivilegedControlRegister(index)	Returns true if the index corresponds to a privileged control register, that is, if index is in [0,31].

Table 25: Support functions for control register access

Functions are also provided for checking the validity of particular control registers, see *Table 26*. The invalid cases are described in *Volume 1, Chapter 15: Control registers*.

Function	Description
IsInvalidPC(spc)	Returns true if the value of spc is invalid when interpreted as a program counter (PC)
IsInvalidSR(ssr)	Returns true if the value of ssr is invalid when interpreted as a status register (SR)

Table 26: Support functions for control register validity checking

1.10 Configuration register model

The array of configuration registers is denoted CFG. A function called ReadConfigurationRegister is provided to read configuration registers. The assignment:

```
result ← ReadConfigurationRegister(index);
```

is equivalent to:

```
result ← CFG[index];
```

A procedure called WriteConfigurationRegister is provided to write configuration registers. The statement:

```
WriteConfigurationRegister(index, value);
```

is equivalent to:

```
CFG[index] ← value;
```

A function is used in the instruction specifications to determine undefined configuration registers, see *Table 27*.

Function	Description
IsUndefinedConfigurationRegister(index)	Returns true if the index corresponds to an undefined configuration register.

Table 27: Support functions for configuration register access

1.11 Floating-point model

The floating-point specification is abstracted using functions to hide the low-level details. Additional information is provided in a tabular form to describe special and exceptional cases. *Volume 1, Chapter 8: SHmedia floating-point* provides a textual description of floating-point operation.

1.11.1 Functions to access SR and FPSCR

The floating-point instruction specifications use a function notation to access SR and FPSCR state. The used functions are described in *Table 28*.

Function	Description
FpuIsDisabled(SR)	True if SR.FD is 1, otherwise false
FpuFlagI(FPSCR)	True if FPSCR.FLAG.I (sticky flag for inexact) is 1, otherwise false
FpuFlagU(FPSCR)	True if FPSCR.FLAG.U (sticky flag for underflow) is 1, otherwise false
FpuFlagO(FPSCR)	True if FPSCR.FLAG.O (sticky flag for overflow) is 1, otherwise false
FpuFlagZ(FPSCR)	True if FPSCR.FLAG.Z (sticky flag for divide by zero) is 1, otherwise false
FpuFlagV(FPSCR)	True if FPSCR.FLAG.V (sticky flag for invalid) is 1, otherwise false
FpuCauseI(FPSCR)	True if FPSCR.CAUSE.I (cause flag for inexact) is 1, otherwise false
FpuCauseU(FPSCR)	True if FPSCR.CAUSE.U (cause flag for underflow) is 1, otherwise false
FpuCauseO(FPSCR)	True if FPSCR.CAUSE.O (cause flag for overflow) is 1, otherwise false
FpuCauseZ(FPSCR)	True if FPSCR.CAUSE.Z (cause flag for divide by zero) is 1, otherwise false
FpuCauseV(FPSCR)	True if FPSCR.CAUSE.V (cause flag for invalid) is 1, otherwise false
FpuCauseE(FPSCR)	True if FPSCR.CAUSE.E (cause flag for FPU error) is 1, otherwise false
FpuEnableI(FPSCR)	True if FPSCR.ENABLE.I (exception enable for inexact) is 1, otherwise false
FpuEnableU(FPSCR)	True if FPSCR.ENABLE.U (exception enable for underflow) is 1, otherwise false
FpuEnableO(FPSCR)	True if FPSCR.ENABLE.O (exception enable for overflow) is 1, otherwise false
FpuEnableZ(FPSCR)	True if FPSCR.ENABLE.Z (exception enable for divide by zero) is 1, otherwise false

Table 28: SR and FPSCR access

Function	Description
FpuEnableV(FPSCR)	True if FPSCR.ENABLE.V (exception enable for invalid) is 1, otherwise false

Table 28: SR and FPSCR access

1.11.2 Functions to model floating-point behavior

Functions are used to model almost all of the floating-point behavior. Each function is associated with a list of results and a list of parameters. The functions encapsulate the computation associated with the instruction. This includes handling of input denormalized values, special case detection, exceptional cases and the floating-point arithmetic.

The following tables summarize the functions used by each instruction. The table shows how the parameters are interpreted and how the results are computed. The $n^{th.}$ parameter is denoted as Pn and the $n^{th.}$ result as RESn.

The parameters and results of these functions are all modeled as integer values. For floating-point parameters and results, these values are integer bit-patterns representing the IEEE754 formats. Multi-value results are used to return two results: the computed result and a new value for FPSCR. If the new value of FPSCR causes an exception to be raised, then the destination register will not be updated with the computed result.

Instruction	Function	RES0	RES1	P0, P1	P2
FADD.S	FADD_S	Single result of (P0 $+_{IEEE754}$ P1)	New FPSCR	Single	Old FPSCR
FADD.D	FADD_D	Double result of (P0 $+_{IEEE754}$ P1)	New FPSCR	Double	Old FPSCR
FSUB.S	FSUB_S	Single result of (P0 $-_{IEEE754}$ P1)	New FPSCR	Single	Old FPSCR
FSUB.D	FSUB_D	Double result of (P0 $-_{IEEE754}$ P1)	New FPSCR	Double	Old FPSCR
FMUL.S	FMUL_S	Single result of (P0 $\times_{IEEE754}$ P1)	New FPSCR	Single	Old FPSCR
FMUL.D	FMUL_D	Double result of (P0 $\times_{IEEE754}$ P1)	New FPSCR	Double	Old FPSCR
FDIV.S	FDIV_S	Single result of (P0 $/_{IEEE754}$ P1)	New FPSCR	Single	Old FPSCR
FDIV.D	FDIV_D	Double result of (P0 $/_{IEEE754}$ P1)	New FPSCR	Double	Old FPSCR

Table 29: Floating-point dyadic arithmetic

Instruction	Function	RES0	RES1	P0	P1
FABS.S	FABS_S	Single result of absolute P0	(not used)	Single	Old FPSCR
FABS.D	FABS_D	Double result of absolute P0	(not used)	Double	Old FPSCR
FNEG.S	FNEG_S	Single result of negating P0	(not used)	Single	Old FPSCR
FNEG.D	FNEG_D	Double result of negating of P0	(not used)	Double	Old FPSCR
FSQRT.S	FSQRT_S	Single result of $_{IEEE754}\sqrt{}$P0	New FPSCR	Single	Old FPSCR
FSQRT.D	FSQRT_D	Double result of $_{IEEE754}\sqrt{}$P0	New FPSCR	Double	Old FPSCR

Table 30: Floating-point monadic arithmetic

Instruction	Function	RES0	RES1	P0, P1	P2
FCMPEQ.S	FCMPEQ_S	Boolean result of (P0 $=_{IEEE754}$ P1)	New FPSCR	Single	Old FPSCR
FCMPEQ.D	FCMPEQ_D	Boolean result of (P0 $=_{IEEE754}$ P1)	New FPSCR	Double	Old FPSCR
FCMPGT.S	FCMPGT_S	Boolean result of (P0 $>_{IEEE754}$ P1)	New FPSCR	Single	Old FPSCR
FCMPGT.D	FCMPGT_D	Boolean result of (P0 $>_{IEEE754}$ P1)	New FPSCR	Double	Old FPSCR
FCMPGE.S	FCMPGE_S	Boolean result of (P0 $\geq_{IEEE754}$ P1)	New FPSCR	Single	Old FPSCR
FCMPGE.D	FCMPGE_D	Boolean result of (P0 $\geq_{IEEE754}$ P1)	New FPSCR	Double	Old FPSCR
FCMPUN.S	FCMPUN_S	Boolean result of (P0 $?_{IEEE754}$ P1)	New FPSCR	Single	Old FPSCR
FCMPUN.D	FCMPUN_D	Boolean result of (P0 $?_{IEEE754}$ P1)	New FPSCR	Double	Old FPSCR

Table 31: Floating-point comparisons

Instruction	Function	RES0	RES1	P0	P1
FCNV.SD	FCNV_SD	P0 is converted to double result	New FPSCR	Single	Old FPSCR
FCNV.DS	FCNV_DS	P0 is converted to single result	New FPSCR	Double	Old FPSCR
FTRC.SL	FTRC_SL	P0 is converted to signed 32-bit integer result	New FPSCR	Single	Old FPSCR

Table 32: Floating-point conversions

Instruction	Function	RES0	RES1	P0	P1
FTRC.DL	FTRC_DL	P0 is converted to signed 32-bit integer result	New FPSCR	Double	Old FPSCR
FTRC.SQ	FTRC_SQ	P0 is converted to signed 64-bit integer result	New FPSCR	Single	Old FPSCR
FTRC.DQ	FTRC_DQ	P0 is converted to signed 64-bit integer result	New FPSCR	Double	Old FPSCR
FLOAT.LS	FLOAT_LS	P0 is converted to single result	New FPSCR	32-bit int	Old FPSCR
FLOAT.LD	FLOAT_LD	P0 is converted to double result	New FPSCR	32-bit int	Old FPSCR
FLOAT.QS	FLOAT_QS	P0 is converted to single result	New FPSCR	64-bit int	Old FPSCR
FLOAT.QD	FLOAT_QD	P0 is converted to double result	New FPSCR	64-bit int	Old FPSCR

Table 32: Floating-point conversions

Instruction	Function	RES0	RES1	P0, P1, P2	P3
FMAC.S	FMAC_S	Single result of fused (P0 × P1) + P2	New FPSCR	Single	Old FPSCR

Table 33: Floating-point multiply-accumulate

Instruction	Function	RES0	RES1	P0	P1	P2
FIPR.S	FIPR_S	Single result of inner product of P0 with P1	New FPSCR	Array of 4 singles	Array of 4 singles	Old FPSCR
FTRV.S	FTRV_S	Array of 4 single results of matrix transform of P0 with P1	New FPSCR	Array of 16 singles	Array of 4 singles	Old FPSCR

Table 34: Special-purpose floating-point dyadic arithmetic

Instruction	Function	RES0	RES1	P0	P1
FCOSA.S	FCOSA_S	Single result approximating cosine of P0	New FPSCR	32-bit int	Old FPSCR
FSINA.S	FSINA_S	Single result approximating sine of P0	New FPSCR	32-bit int	Old FPSCR

Table 35: Special-purpose floating-point monadic arithmetic

Instruction	Function	RES0	RES1	P0	P1
FSRRA.S	FSRRA_S	Single result approximating reciprocal square root of P0	New FPSCR	Single	Old FPSCR

Table 35: Special-purpose floating-point monadic arithmetic

1.11.3 Floating-point special cases and exceptions

A special-case table is provided for each floating-point instruction that is considered an operation and has at least one input that is interpreted as a floating-point value. This table enumerates all different possible combinations of input values and the results returned by the instruction in the absence of an exception being raised.

Each cell entry in the table describes the result returned for a particular combination of floating-point inputs. If the combination of inputs is sufficiently qualified to indicate a specific result, then its value is quoted in the cell. If they are not sufficiently qualified, the name of the appropriate operation is entered in the cell. If the cell contains 'n/a' then this indicates that an exception is always raised for that combination of inputs and that the implementation does not associate any value with the result.

1.12 Abstract sequential model

Instructions are specified using an abstract sequential model to show the effects of each instruction on the architectural state of the machine. In this abstract model, each instruction executes completely sequentially with respect to other instructions. This means that all actions associated with one instruction are completed before any actions associated with the next instruction are started.

Implementations will generally make substantial optimizations over this abstract model. For typical well-disciplined instruction sequences these effects will not be architecturally visible. However, a fuller description of the behavior in other cases is defined by the text of the architecture manual.

If ISA is 0, the instruction is executed in SHcompact mode as described in *Volume 3 Chapter 1: SHcompact specification*. Otherwise, the instruction is executed in SHmedia mode.

The steps associated with executing each SHmedia instruction are:

1 Check for asynchronous events, such as interrupt or reset, and initiate handling if required.

2 Check the current program counter (PC) for instruction address exceptions, and initiate handling if required. Instruction address exceptions include instruction TLB miss, instruction protection violation and instruction address error.

3 Fetch the instruction bytes from the address in memory, as indicated by the current PC. For SHmedia, 4 bytes need to be fetched for each instruction.

4 Calculate the default value of the next program counter (PC') assuming sequential execution. For SHmedia, PC' is PC+4.

5 Decode and execute the instruction. This includes checks for synchronous events, such as exceptions and panics, and initiation of handling if required. The execution of an instruction can change PC' to achieve a branch.

6 If the value of PC' is outside of the implemented part of the effective address space, then the behavior becomes architecturally undefined.

7 Set the current program counter (PC) to the value of the next program counter (PC').

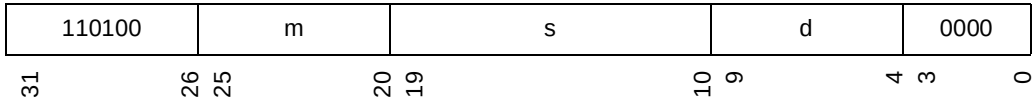
The actions associated with the handling of asynchronous and synchronous events are described in *Volume 1, Chapter 16: Event handling*. The actions required by step *5* depend on the instruction, and are specified by the instruction specification for that instruction. Step *6* specifies the behavior for PC overflow. This is described further in *Volume 1, Chapter 3: Data representation*.

1.13 Example instructions

1.13.1 Integer add immediate

An example specification for this instruction is shown below.

ADDI Rm, imm, Rd



source1 ← SignExtend$_{64}$(R_m);
imm ← SignExtend$_{10}$(s);
result ← source1 + imm;
R_d ← Register(result);

The top half of this figure shows the assembly syntax and the binary encoding of the instruction. Particular fields within the encoding are identified by single characters. The interpretation associated with these characters is given in *Chapter 4: SHmedia instructions on page 57*. The opcode field, and any extension field, contain the literal encoding values associated with that instruction. Reserved fields must be encoded with the literal value given in the figure. Operand fields contain register designators or immediate constants.

The lower half of this figure specifies the effects of the execution of the instruction on the architectural state of the machine. The specification statements are organized into 3 stages as follows:

1 The first 2 statements read all required source information:

source1 ← SignExtend$_{64}$(R_m);
imm ← SignExtend$_{10}$(s);

The first statement reads the value of the R_m register, interprets it as a signed 64-bit integer value and assigns this to a temporary integer called ‘source1’. The second statement reads the value of s, interprets it as a signed 10-bit integer value and assigns this to a temporary integer called ‘imm’. The name ‘imm’ corresponds to the name of the immediate used in the assembly syntax.

2 The next statement implements the addition:

result ← source1 + imm;

This statement does not refer to any architectural state. It adds the 2 integers 'source1' and 'imm' together, and assigns the result to a temporary integer called 'result'. Note that since this is a conventional mathematical addition, the result can contain more significant bits of information than the sources.

3 The final statement updates the architectural state:

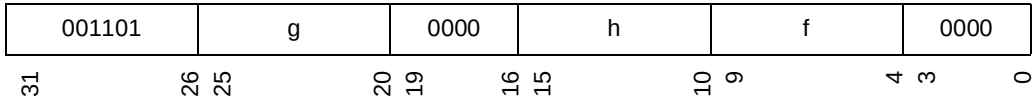
R_d ← Register(result);

The integer 'result' is converted back to the range of a register value, discarding any redundant higher bits, and assigned to the R_d register.

1.13.2 Floating-point single-precision add

An example specification for this instruction is shown below.

FADD.S FRg, FRh, FRf




```
sr ← ZeroExtend64(SR);
fps ← ZeroExtend32(FPSCR);
source1 ← FloatValue32(FRg);
source2 ← FloatValue32(FRh);
IF (FpuIsDisabled(sr))
   THROW FPUDIS;
result, fps ← FADD_S(source1, source2, fps);
IF (FpuEnableV(fps) AND FpuCauseV(fps))
   THROW FPUEXC, fps;
IF (FpuCauseE(fps))
   THROW FPUEXC, fps;
IF ((FpuEnableI(fps) OR FpuEnableO(fps)) OR FpuEnableU(fps))
   THROW FPUEXC, fps;
FRf ← FloatRegister32(result);
FPSCR ← Register(fps);
```

The specification statements are organized as follows:

1 Read all required source information:

```
sr ← ZeroExtend64(SR);
fps ← ZeroExtend32(FPSCR);
source1 ← FloatValue32(FRg);
source2 ← FloatValue32(FRh);
```

2 Execute the instruction:

```
IF (FpuIsDisabled(sr))
    THROW FPUDIS;
result, fps ← FADD_S(source1, source2, fps);
IF (FpuEnableV(fps) AND FpuCauseV(fps))
    THROW FPUEXC, fps;
IF (FpuCauseE(fps))
    THROW FPUEXC, fps;
IF ((FpuEnableI(fps) OR FpuEnableO(fps)) OR FpuEnableU(fps))
    THROW FPUEXC, fps;
```

The behavior of the floating-point single-precision addition is modelled by the FADD_S procedure. This procedure is given the two source operands and the current value of FPSCR, and calculates the result and the new value of FPSCR. It is responsible for detecting special cases and exceptions, and setting the result and new FPSCR values accordingly.

This instruction contains exception cases. These are detected by IF statements and are raised by THROW statements. When a THROW statement is executed, no further statements from the specification are processed. Note that when an exception is detected the specification makes no updates to the architectural state. Instead, a handler is launched for the exception as described in *Volume 1, Chapter 16: Event handling*. The THROW statement includes arguments to specify the kind of exception and any necessary parameters for that exception. For an FPUEXC exception, the THROW statement includes an updated value of 'fps' which the exception handler uses to initialize FPSCR during the launch sequence.

3 Update the architectural state:

```
FRf ← FloatRegister32(result);
FPSCR ← Register(fps);
```

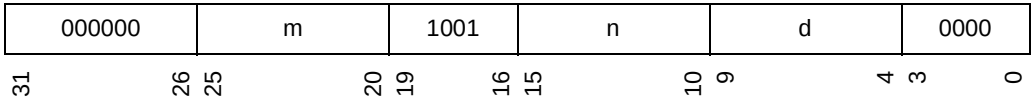
SuperH

2 SHmedia instruction set

2.1 Alphabetical list of instructions

ADD Rm, Rn, Rd

ADD Rm, Rn, Rd



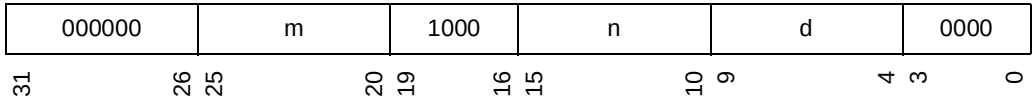
source1 ← SignExtend$_{64}$(R_m);
source2 ← SignExtend$_{64}$(R_n);
result ← source1 + source2;
R_d ← Register(result);

Description:

This instruction adds R_m to R_n and places the result in R_d.

ADD.L Rm, Rn, Rd

ADD.L Rm, Rn, Rd



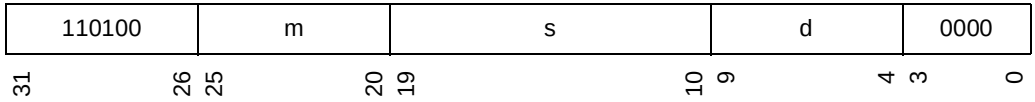

```
source1 ← SignExtend32(Rm);
source2 ← SignExtend32(Rn);
result ← SignExtend32(source1 + source2);
Rd ← Register(result);
```

Description:

This instruction adds the lowest 32 bits of R_m to the lowest 32 bits of R_n and places the sign-extended 32-bit result in R_d. The highest 32 bits of R_m and the highest 32 bits of R_n are ignored.

ADDI Rm, imm, Rd

ADDI Rm, imm, Rd




```
source1 ← SignExtend64(Rm);
imm ← SignExtend10(s);
result ← source1 + imm;
Rd ← Register(result);
```

Description:

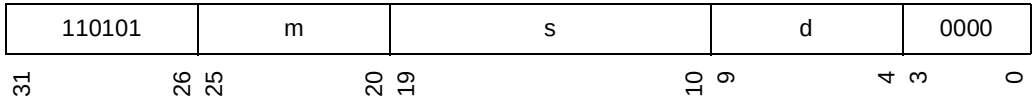
This instruction adds R_m to the sign-extended 10-bit immediate s and places the result in R_d.

Notes:

The 'imm' in the assembly syntax represents the immediate s after sign extension.

ADDI.L Rm, imm, Rd

ADDI.L Rm, imm, Rd



source1 ← SignExtend$_{32}$(R_m);
imm ← SignExtend$_{10}$(s);
result ← SignExtend$_{32}$(source1 + imm);
R_d ← Register(result);

Description:

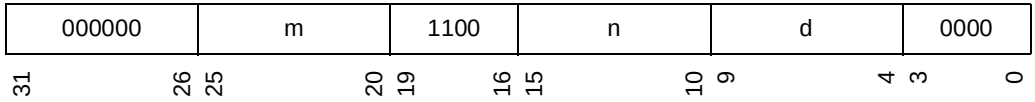
This instruction adds the lowest 32 bits of R_m to the sign-extended 10-bit immediate s, and places the sign-extended 32-bit result in R_d. The highest 32 bits of R_m are ignored.

Notes:

The 'imm' in the assembly syntax represents the immediate s after sign extension.

ADDZ.L Rm, Rn, Rd

ADDZ.L Rm, Rn, Rd



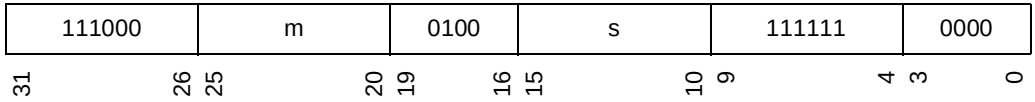
source1 ← ZeroExtend$_{32}$(R_m);
source2 ← ZeroExtend$_{32}$(R_n);
result ← ZeroExtend$_{32}$(source1 + source2);
R_d ← Register(result);

Description:

This instruction adds the lowest 32 bits of R_m to the lowest 32 bits of R_n and places the zero-extended 32-bit result in R_d. The highest 32 bits of R_m and the highest 32 bits of R_n are ignored.

ALLOCO Rm, disp

ALLOCO Rm, disp




```
base ← ZeroExtend64(Rm);
disp ← SignExtend6(s) << 5;
address ← ZeroExtend64(base + disp);
IF (MalformedAddress(address))
   THROW WADDERR, address;
IF (MMU() AND DataAccessMiss(address))
   THROW WTLBMISS, address;
IF (MMU() AND WriteProhibited(address))
   THROW WRITEPROT, address;
ALLOCO(address);
```

Description:

This instruction is used to request allocation of an operand cache block for a specified effective address. It provides a hint to the implementation that it is not necessary to retrieve the data of this operand cache block from memory. It is implementation-specific as to whether the memory access will occur.

The effective address is calculated by adding R_m to the sign-extended 6-bit immediate s multiplied by 32.The scaling factor is fixed at 32 regardless of the cache block size. There is no misalignment check on this instruction, and the calculated effective address can be any byte address. The calculated effective address is automatically aligned downwards to the nearest exact multiple of the cache block size. The effective address identifies a surrounding block of memory, which starts at an address aligned to the cache block size and has a size equal to the cache block size. The cache block size is implementation dependent.

ALLOCO checks for address error, translation miss and protection exception cases.

The value of each location in the memory block targeted by an ALLOCO becomes architecturally undefined. Programs must not rely on these values. For compatibility with other implementations, software must exercise care when using ALLOCO.

Possible exceptions:

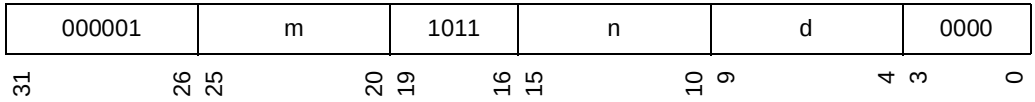
WADDERR, WTLBMISS, WRITEPROT

Notes:

The ‘disp’ in the assembly syntax represents the immediate s after sign extension and scaling.

AND Rm, Rn, Rd

AND Rm, Rn, Rd



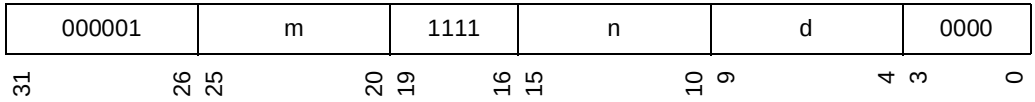
source1 ← SignExtend$_{64}$(R_m);
source2 ← SignExtend$_{64}$(R_n);
result ← source1 ∧ source2;
R_d ← Register(result);

Description:

This instruction performs a bitwise AND of R_m with R_n and places the result in R_d.

ANDC Rm, Rn, Rd

ANDC Rm, Rn, Rd



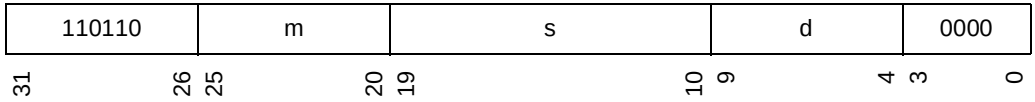
source1 ← SignExtend$_{64}$(R_m);
source2 ← SignExtend$_{64}$(R_n);
result ← source1 ∧ (~ source2);
R_d ← Register(result);

Description:

This instruction performs a bitwise AND of R_m with the bitwise NOT of R_n and places the result in R_d.

ANDI Rm, imm, Rd

ANDI Rm, imm, Rd




```
source1 ← SignExtend64(Rm);
imm ← SignExtend10(s);
result ← source1 ∧ imm;
Rd ← Register(result);
```

Description:

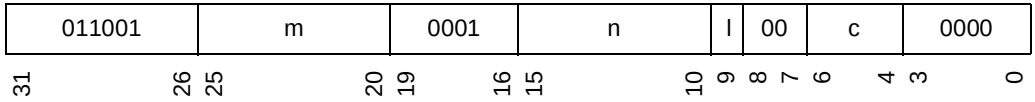
This instruction performs a bitwise AND of R_m with the sign-extended 10-bit immediate s and places the result in R_d.

Notes:

The 'imm' in the assembly syntax represents the immediate s after sign extension.

BEQ Rm, Rn, TRc

BEQ Rm, Rn, TRc




```
newpc ← ZeroExtend64(PC');
source1 ← SignExtend64(Rm);
source2 ← SignExtend64(Rn);
target ← ZeroExtend64(TRc);
IF (source1 = source2)
   newpc ← target ∧ (~ 0x3);
PC' ← Register(newpc);
```

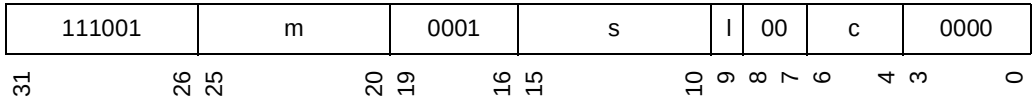
Description:

This instruction copies the value of the target address held in TR_c to the PC if the value in R_m equals the value in R_n. The lowest 2 bits of the target address are masked to zero in this copy.

The encoding contains a single bit, labeled l, which is used to indicate whether it is likely (1) or unlikely (0) that the branch will be taken. This bit is encoded as 1 if the instruction mnemonic is 'BEQ' or 'BEQ/L', or as 0 if the mnemonic is 'BEQ/U'.

BEQI Rm, imm, TRc

BEQI Rm, imm, TRc




```
newpc ← ZeroExtend64(PC');
source1 ← SignExtend64(Rm);
imm ← SignExtend6(s);
target ← ZeroExtend64(TRc);
IF (source1 = imm)
   newpc ← target ∧ (~ 0x3);
PC' ← Register(newpc);
```

Description:

This instruction copies the value of the target address held in TR_c to the PC if R_m equals the sign-extended 6-bit immediate s. The lowest 2 bits of the target address are masked to zero in this copy.

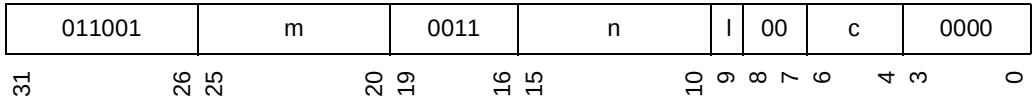
The encoding contains a single bit, labeled l, which is used to indicate whether it is likely (1) or unlikely (0) that the branch will be taken. This bit is encoded as 1 if the instruction mnemonic is 'BEQI' or 'BEQI/L', or as 0 if the mnemonic is 'BEQI/U'.

Notes:

The 'imm' in the assembly syntax represents the immediate s after sign extension.

BGE Rm, Rn, TRc

BGE Rm, Rn, TRc




```
newpc ← ZeroExtend64(PC');
source1 ← SignExtend64(Rm);
source2 ← SignExtend64(Rn);
target ← ZeroExtend64(TRc);
IF (source1 ≥ source2)
   newpc ← target ∧ (~ 0x3);
PC' ← Register(newpc);
```

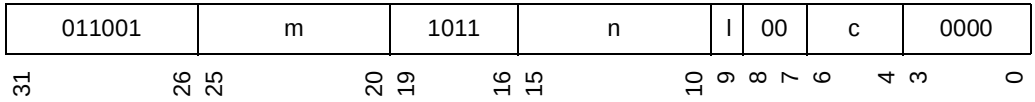
Description:

This instruction copies the value of the target address held in TR_c to the PC if the signed value in R_m is greater than or equal to the signed value in R_n. The lowest 2 bits of the target address are masked to zero in this copy.

The encoding contains a single bit, labeled l, which is used to indicate whether it is likely (1) or unlikely (0) that the branch will be taken. This bit is encoded as 1 if the instruction mnemonic is 'BGE' or 'BGE/L', or as 0 if the mnemonic is 'BGE/U'.

BGEU Rm, Rn, TRc

BGEU Rm, Rn, TRc




```
newpc ← ZeroExtend64(PC');
source1 ← ZeroExtend64(Rm);
source2 ← ZeroExtend64(Rn);
target ← ZeroExtend64(TRc);
IF (source1 ≥ source2)
   newpc ← target ∧ (~ 0x3);
PC' ← Register(newpc);
```

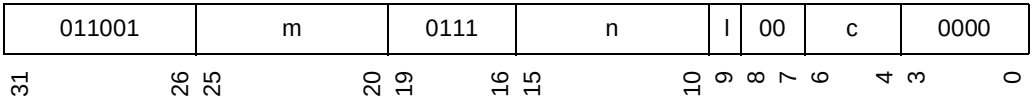
Description:

This instruction copies the value of the target address held in TRc to the PC if the unsigned value in R_m is greater than or equal to the unsigned value in R_n. The lowest 2 bits of the target address are masked to zero in this copy.

The encoding contains a single bit, labeled l, which is used to indicate whether it is likely (1) or unlikely (0) that the branch will be taken. This bit is encoded as 1 if the instruction mnemonic is 'BGEU' or 'BGEU/L', or as 0 if the mnemonic is 'BGEU/U'.

BGT Rm, Rn, TRc

BGT Rm, Rn, TRc




```
newpc ← ZeroExtend64(PC');
source1 ← SignExtend64(Rm);
source2 ← SignExtend64(Rn);
target ← ZeroExtend64(TRc);
IF (source1 > source2)
   newpc ← target ∧ (~ 0x3);
PC' ← Register(newpc);
```

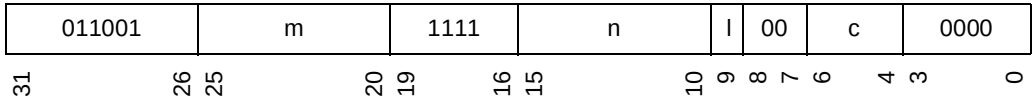
Description:

This instruction copies the value of the target address held in TR_c to the PC if the signed value in R_m is greater than the signed value in R_n. The lowest 2 bits of the target address are masked to zero in this copy.

The encoding contains a single bit, labeled l, which is used to indicate whether it is likely (1) or unlikely (0) that the branch will be taken. This bit is encoded as 1 if the instruction mnemonic is 'BGT' or 'BGT/L', or as 0 if the mnemonic is 'BGT/U'.

BGTU Rm, Rn, TRc

BGTU Rm, Rn, TRc




```
newpc ← ZeroExtend64(PC');
source1 ← ZeroExtend64(Rm);
source2 ← ZeroExtend64(Rn);
target ← ZeroExtend64(TRc);
IF (source1 > source2)
   newpc ← target ∧ (~ 0x3);
PC' ← Register(newpc);
```

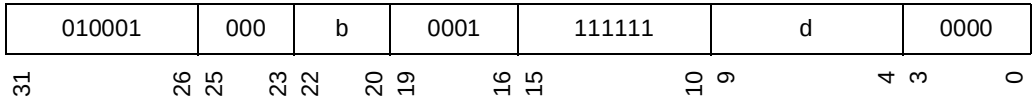
Description:

This instruction copies the value of the target address held in TR_c to the PC if the unsigned value in R_m is greater than the unsigned value in R_n. The lowest 2 bits of the target address are masked to zero in this copy.

The encoding contains a single bit, labeled l, which is used to indicate whether it is likely (1) or unlikely (0) that the branch will be taken. This bit is encoded as 1 if the instruction mnemonic is 'BGTU' or 'BGTU/L', or as 0 if the mnemonic is 'BGTU/U'.

BLINK TRb, Rd

BLINK TRb, Rd




```
pc ← ZeroExtend64(PC);
isa ← ZeroExtend1(ISA);
target ← ZeroExtend64(TRb);
address ← pc + 4;
IF (MalformedAddress(address))
   link ← UNDEFINED;
ELSE
   link ← address + isa;
isa ← target ∧ 0x1;
newpc ← target ∧ (~ 0x1);
Rd ← Register(link);
PC' ← Register(newpc);
ISA ← Bit(isa);
```

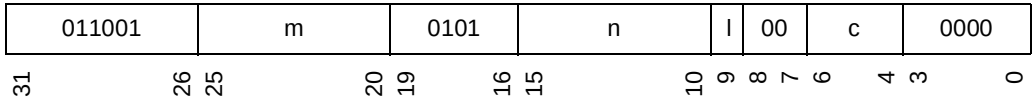
Description:

This instruction reads the target address held in TR_b, clears the least significant bit, and copies this value to the PC. Bit 0 of TR_b gives the new value of the ISA mode for the next instruction (0 indicates SHcompact and 1 indicates SHmedia). The address of the following instruction, with the lowest bit set to 1 to indicate SHmedia, is placed in R_d.

BLINK calculates PC+4 to determine the address of the following instruction and it is possible for this calculation to give a malformed address. The value placed in R_d will then be architecturally undefined. This case corresponds to program counter overflow as described in *Volume 1, Chapter 3: Data representation*. When program counter overflow occurs for any instruction, the behavior becomes architecturally undefined. For BLINK the setting of R_d to an architecturally undefined value is one aspect of this architecturally undefined behavior, but further undefined behavior is also possible.

BNE Rm, Rn, TRc

BNE Rm, Rn, TRc




```
newpc ← ZeroExtend64(PC');
source1 ← SignExtend64(Rm);
source2 ← SignExtend64(Rn);
target ← ZeroExtend64(TRc);
IF (source1 ≠ source2)
   newpc ← target ∧ (~ 0x3);
PC' ← Register(newpc);
```

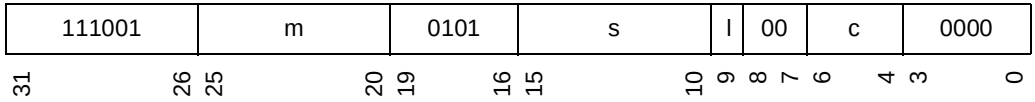
Description:

This instruction copies the value of the target address held in TR_c to the PC if the value in R_m does not equal the value in R_n. The lowest 2 bits of the target address are masked to zero in this copy.

The encoding contains a single bit, labeled l, which is used to indicate whether it is likely (1) or unlikely (0) that the branch will be taken. This bit is encoded as 1 if the instruction mnemonic is 'BNE' or 'BNE/L', or as 0 if the mnemonic is 'BNE/U'.

BNEI Rm, imm, TRc

BNEI Rm, imm, TRc




```
newpc ← ZeroExtend64(PC');
source1 ← SignExtend64(Rm);
imm ← SignExtend6(s);
target ← ZeroExtend64(TRc);
IF (source1 ≠ imm)
   newpc ← target ∧ (~ 0x3);
PC' ← Register(newpc);
```

Description:

This instruction copies the value of the target address held in TR_c to the PC if R_m does not equal the sign-extended 6-bit immediate s. The lowest 2 bits of the target address are masked to zero in this copy.

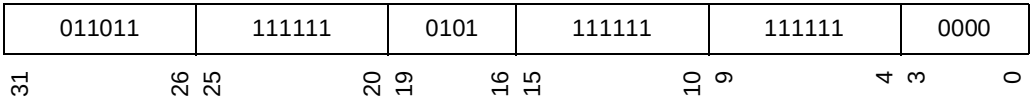
The encoding contains a single bit, labeled l, which is used to indicate whether it is likely (1) or unlikely (0) that the branch will be taken. This bit is encoded as 1 if the instruction mnemonic is 'BNEI' or 'BNEI/L', or as 0 if the mnemonic is 'BNEI/U'.

Notes:

The 'imm' in the assembly syntax represents the immediate s after sign extension.

BRK

BRK




```
THROW BREAK;
```

Description:

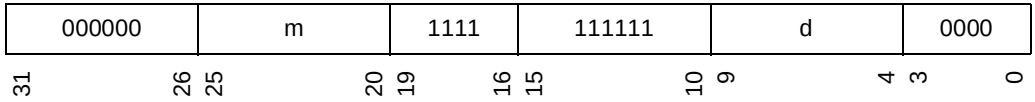
This instruction causes a pre-execution break exception. The BRK instruction is typically reserved for use by the debugger.

Possible exceptions:

BREAK

BYTEREV Rm, Rd

BYTEREV Rm, Rd



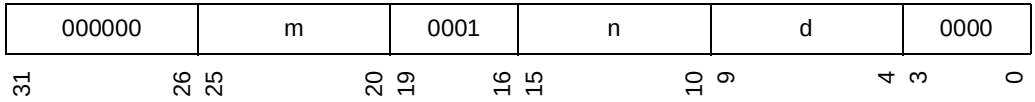

```
source ← ZeroExtend64(Rm);
result ← 0;
REPEAT i FROM 0 FOR 8
{
   result ← (result << 8) ∨ (source ∧ 0xff);
   source ← source >> 8;
}
Rd ← Register(result);
```

Description:

This instruction reverses the 8 bytes contained in R_m and places the result in R_d.

CMPEQ Rm, Rn, Rd

CMPEQ Rm, Rn, Rd



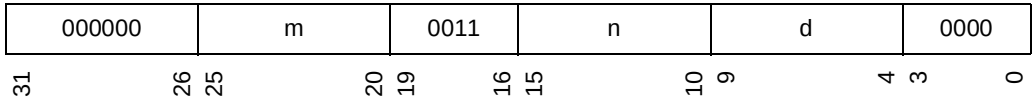
source1 ← SignExtend$_{64}$(R_m);
source2 ← SignExtend$_{64}$(R_n);
result ← INT (source1 = source2);
R_d ← Register(result);

Description:

This instruction sets R_d to 1 if the value of R_m is equal to the value of R_n, otherwise it sets R_d to 0.

CMPGT Rm, Rn, Rd

CMPGT Rm, Rn, Rd



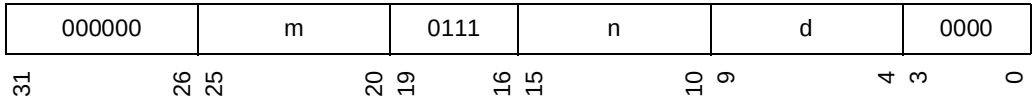
source1 ← SignExtend$_{64}$(R_m);
source2 ← SignExtend$_{64}$(R_n);
result ← INT (source1 > source2);
R_d ← Register(result);

Description:

This instruction sets R_d to 1 if the signed value of R_m is greater than the signed value of R_n, otherwise it sets R_d to 0.

CMPGTU Rm, Rn, Rd

CMPGTU Rm, Rn, Rd



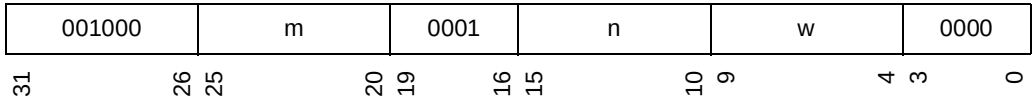
source1 ← ZeroExtend$_{64}$(R_m);
source2 ← ZeroExtend$_{64}$(R_n);
result ← INT (source1 > source2);
R_d ← Register(result);

Description:

This instruction sets R_d to 1 if the unsigned value of R_m is greater than the unsigned value of R_n, otherwise it sets R_d to 0.

CMVEQ Rm, Rn, Rw

CMVEQ Rm, Rn, Rw




```
source1 ← SignExtend64(Rm);
source2 ← SignExtend64(Rn);
source3_result ← ZeroExtend64(Rw);
IF (source1 = 0)
   source3_result ← source2;
Rw ← Register(source3_result);
```

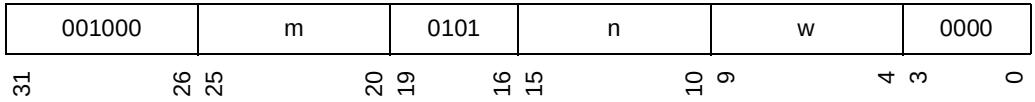
Description:

This instruction copies R_n to R_w if the value of R_m is 0, otherwise R_w is not changed.

The mnemonic CMVEQ stands for a 'Conditional MoVe if EQual to zero'.

CMVNE Rm, Rn, Rw

CMVNE Rm, Rn, Rw




```
source1 ← SignExtend64(Rm);
source2 ← SignExtend64(Rn);
source3_result ← ZeroExtend64(Rw);
IF (source1 ≠ 0)
   source3_result ← source2;
Rw ← Register(source3_result);
```

Description:

This instruction copies R_n to R_w if the value of R_m is not 0, otherwise R_w is not changed.

The mnemonic CMVEQ stands for a 'Conditional MoVe if Not EQual to zero'.

FABS.D DRg, DRf

FABS.D DRg, DRf

000110	g	0001	g	f	0000
31 26	25 20	19 16	15 10	9 4	3 0

```
sr ← ZeroExtend64(SR);
source ← FloatValue64(DRg);
IF (FpuIsDisabled(sr))
    THROW FPUDIS;
result ← FABS_D(source);
DRf ← FloatRegister64(result);
```

Description:

This floating-point instruction computes the absolute value of a double-precision floating-point number. It reads DR_g, clears the sign bit and places the result in DR_f.

This instruction is not considered an arithmetic operation, and it does not signal invalid operations. There are no special floating-point cases associated with this instruction.

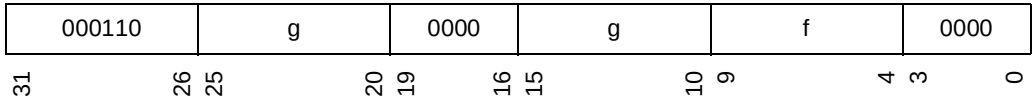
Both source operand fields must be encoded with the source register designator g.

Possible exceptions:

FPUDIS

FABS.S FRg, FRf

FABS.S FRg, FRf




```
sr ← ZeroExtend64(SR);
source ← FloatValue32(FRg);
IF (FpuIsDisabled(sr))
   THROW FPUDIS;
result ← FABS_S(source);
FRf ← FloatRegister32(result);
```

Description:

This floating-point instruction computes the absolute value of a single-precision floating-point number. It reads FR_g, clears the sign bit and places the result in FR_f.

This instruction is not considered an arithmetic operation, and it does not signal invalid operations. There are no special floating-point cases associated with this instruction.

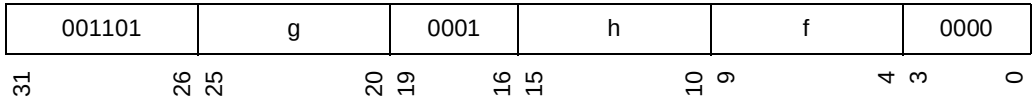
Both source operand fields must be encoded with the source register designator g.

Possible exceptions:

FPUDIS

FADD.D DRg, DRh, DRf

FADD.D DRg, DRh, DRf




```
sr ← ZeroExtend64(SR);
fps ← ZeroExtend32(FPSCR);
source1 ← FloatValue64(DRg);
source2 ← FloatValue64(DRh);
IF (FpuIsDisabled(sr))
   THROW FPUDIS;
result, fps ← FADD_D(source1, source2, fps);
IF (FpuEnableV(fps) AND FpuCauseV(fps))
   THROW FPUEXC, fps;
IF (FpuCauseE(fps))
   THROW FPUEXC, fps;
IF ((FpuEnableI(fps) OR FpuEnableO(fps)) OR FpuEnableU(fps))
   THROW FPUEXC, fps;
DRf ← FloatRegister64(result);
FPSCR ← Register(fps);
```

Description:

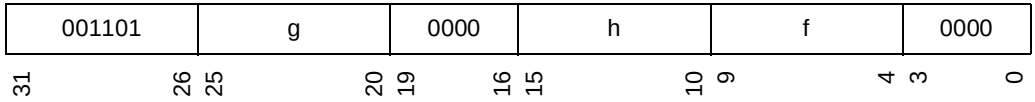
This floating-point instruction performs a double-precision floating-point addition. It adds DR_g to DR_h and places the result in DR_f. The rounding mode is determined by FPSCR.RM.

Possible exceptions:

FPUDIS, FPUEXC

FADD.S FRg, FRh, FRf

FADD.S FRg, FRh, FRf




```
sr ← ZeroExtend64(SR);
fps ← ZeroExtend32(FPSCR);
source1 ← FloatValue32(FRg);
source2 ← FloatValue32(FRh);
IF (FpuIsDisabled(sr))
   THROW FPUDIS;
result, fps ← FADD_S(source1, source2, fps);
IF (FpuEnableV(fps) AND FpuCauseV(fps))
   THROW FPUEXC, fps;
IF (FpuCauseE(fps))
   THROW FPUEXC, fps;
IF ((FpuEnableI(fps) OR FpuEnableO(fps)) OR FpuEnableU(fps))
   THROW FPUEXC, fps;
FRf ← FloatRegister32(result);
FPSCR ← Register(fps);
```

Description:

This floating-point instruction performs a single-precision floating-point addition. It adds FR_g to FR_h and places the result in FR_f. The rounding mode is determined by FPSCR.RM.

Possible exceptions:

FPUDIS, FPUEXC

FADD.S and FADD.D special cases:

When FPSCR.DN is 1, a positive denormalized number is treated as +0 and a negative denormalized number as -0. This flush-to-zero treatment is applied before exception detection and special case handling.

Exceptional conditions are checked in the order given below. Execution of the instruction is terminated once any check detects an exceptional condition.

1 Disabled: an exception is raised if the FPU is disabled.

2 Invalid: an invalid operation is signaled if either input is a signaling NaN, or if the inputs are differently signed infinities.

3 Error: an FPU error is signaled if FPSCR.DN is 0, neither input is a NaN and either input is a denormalized number.

4 Inexact, underflow and overflow: these are checked together and can be signaled in combination. When inexact, underflow or overflow exceptions are requested by the user, an exception is always raised regardless of whether that condition arose.

If the instruction does not raise an exception, a result is generated according to the following table.

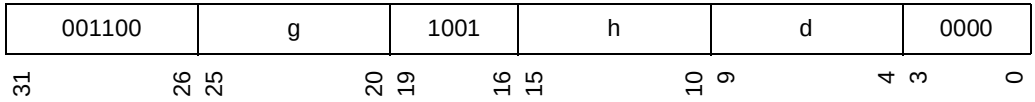
source1 → ↓ source2	+NORM, -NORM	+0	-0	+INF	-INF	+DNRM -DNRM	qNaN	sNaN
+,-NORM	ADD	source2	source2	+INF	-INF	n/a	qNaN	qNaN
+0	source1	+0	+0	+INF	-INF	n/a	qNaN	qNaN
-0	source1	+0	-0	+INF	-INF	n/a	qNaN	qNaN
+INF	+INF	+INF	+INF	+INF	qNaN	n/a	qNaN	qNaN
-INF	-INF	-INF	-INF	qNaN	-INF	n/a	qNaN	qNaN
+, -DNRM	n/a	n/a	n/a	n/a	n/a	n/a	qNaN	qNaN
qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN
sNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN

FPU error is indicated by heavy shading and always raises an exception. Invalid operations are indicated by light shading and raise an exception if enabled. FPU disabled, inexact, underflow and overflow cases are not shown.

The behavior of the normal ‘ADD’ case is described by the IEEE754 specification.

FCMPEQ.D DRg, DRh, Rd

FCMPEQ.D DRg, DRh, Rd




```
sr ← ZeroExtend64(SR);
fps ← ZeroExtend32(FPSCR);
source1 ← FloatValue64(DRg);
source2 ← FloatValue64(DRh);
IF (FpuIsDisabled(sr))
   THROW FPUDIS;
result, fps ← FCMPEQ_D(source1, source2, fps);
IF (FpuEnableV(fps) AND FpuCauseV(fps))
   THROW FPUEXC, fps;
Rd ← Register(result);
FPSCR ← Register(fps);
```

Description:

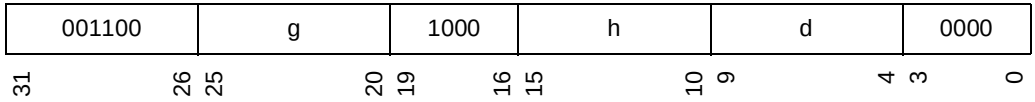
This floating-point instruction performs a double-precision floating-point equality comparison. It sets R_d to 1 if DR_g is equal to DR_h, and otherwise sets R_d to 0.

Possible exceptions:

FPUDIS, FPUEXC

FCMPEQ.S FRg, FRh, Rd

FCMPEQ.S FRg, FRh, Rd




```
sr ← ZeroExtend64(SR);
fps ← ZeroExtend32(FPSCR);
source1 ← FloatValue32(FRg);
source2 ← FloatValue32(FRh);
IF (FpuIsDisabled(sr))
   THROW FPUDIS;
result, fps ← FCMPEQ_S(source1, source2, fps);
IF (FpuEnableV(fps) AND FpuCauseV(fps))
   THROW FPUEXC, fps;
Rd ← Register(result);
FPSCR ← Register(fps);
```

Description:

This floating-point instruction performs a single-precision floating-point equality comparison. It sets R_d to 1 if FR_g is equal to FR_h, and otherwise sets R_d to 0.

Possible exceptions:

FPUDIS, FPUEXC

FCMPEQ.S and FCMPEQ.D special cases:

When FPSCR.DN is 1, a positive denormalized number is treated as +0 and a negative denormalized number as -0. This flush-to-zero treatment is applied before exception detection and special case handling.

Exceptional conditions are checked in the order given below. Execution of the instruction is terminated once any check detects an exceptional condition.

1 Disabled: an exception is raised if the FPU is disabled.

2 Invalid: an invalid operation is signaled if either input is a signaling NaN.

If the instruction does not raise an exception, a result is generated according to the following table.

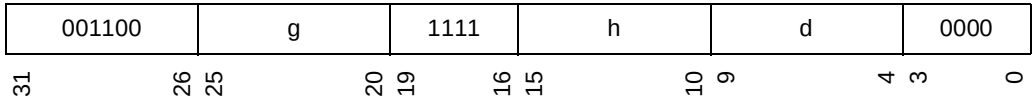
source1 → ↓ source2	+NORM, -NORM	+0	-0	+INF	-INF	+DNRM, -DNRM	qNaN	sNaN
+,-NORM	CMPEQ	false	false	false	false	false	false	false
+0	false	true	true	false	false	false	false	false
-0	false	true	true	false	false	false	false	false
+INF	false	false	false	true	false	false	false	false
-INF	false	false	false	false	true	false	false	false
+, -DNRM	false	false	false	false	false	CMPEQ	false	false
qNaN	false	false	false	false	false	false	false	false
sNaN	false	false	false	false	false	false	false	false

Invalid operations are indicated by light shading and raise an exception if enabled. FPU disabled cases are not shown.

The behavior of the normal 'CMPEQ' case is described by the IEEE754 specification.

FCMPGE.D DRg, DRh, Rd

FCMPGE.D DRg, DRh, Rd




```
sr ← ZeroExtend64(SR);
fps ← ZeroExtend32(FPSCR);
source1 ← FloatValue64(DRg);
source2 ← FloatValue64(DRh);
IF (FpuIsDisabled(sr))
   THROW FPUDIS;
result, fps ← FCMPGE_D(source1, source2, fps);
IF (FpuEnableV(fps) AND FpuCauseV(fps))
   THROW FPUEXC, fps;
Rd ← Register(result);
FPSCR ← Register(fps);
```

Description:

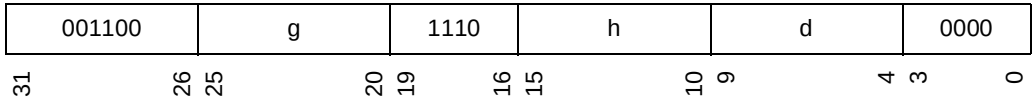
This floating-point instruction performs a double-precision floating-point greater-than-or-equal-to comparison. It sets R_d to 1 if DR_g is greater than or equal to DR_h, and otherwise sets R_d to 0.

Possible exceptions:

FPUDIS, FPUEXC

FCMPGE.S FRg, FRh, Rd

FCMPGE.S FRg, FRh, Rd




```
sr ← ZeroExtend64(SR);
fps ← ZeroExtend32(FPSCR);
source1 ← FloatValue32(FRg);
source2 ← FloatValue32(FRh);
IF (FpuIsDisabled(sr))
   THROW FPUDIS;
result, fps ← FCMPGE_S(source1, source2, fps);
IF (FpuEnableV(fps) AND FpuCauseV(fps))
   THROW FPUEXC, fps;
Rd ← Register(result);
FPSCR ← Register(fps);
```

Description:

This floating-point instruction performs a single-precision floating-point greater-than-or-equal-to comparison. It sets R_d to 1 if FR_g is greater than or equal to FR_h, and otherwise sets R_d to 0.

Possible exceptions:

FPUDIS, FPUEXC

FCMPGE.S and FCMPGE.D special cases:

When FPSCR.DN is 1, a positive denormalized number is treated as +0 and a negative denormalized number as -0. This flush-to-zero treatment is applied before exception detection and special case handling.

Exceptional conditions are checked in the order given below. Execution of the instruction is terminated once any check detects an exceptional condition.

1 Disabled: an exception is raised if the FPU is disabled.

2 Invalid: an invalid operation is signaled if either input is a NaN.

If the instruction does not raise an exception, a result is generated according to the following table.

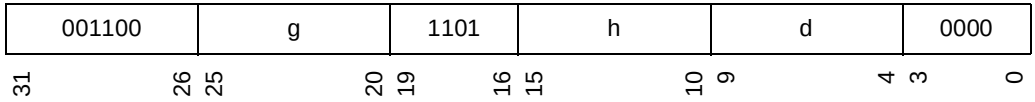
source1 → ↓ source2	+NORM, -NORM	+0	-0	+INF	-INF	+DNRM, -DNRM	qNaN	sNaN
+,-NORM	CMPGE	CMPGE	CMPGE	true	false	CMPGE	false	false
+0	CMPGE	true	false	true	false	CMPGE	false	false
-0	CMPGE	true	true	true	false	CMPGE	false	false
+INF	false	false	false	true	false	false	false	false
-INF	true	true	true	true	true	true	false	false
+, -DNRM	CMPGE	CMPGE	CMPGE	true	false	CMPGE	false	false
qNaN	false	false	false	false	false	false	false	false
sNaN	false	false	false	false	false	false	false	false

Invalid operations are indicated by light shading and raise an exception if enabled. FPU disabled cases are not shown.

The behavior of the normal 'CMPGE' case is described by the IEEE754 specification.

FCMPGT.D DRg, DRh, Rd

FCMPGT.D DRg, DRh, Rd




```
sr ← ZeroExtend64(SR);
fps ← ZeroExtend32(FPSCR);
source1 ← FloatValue64(DRg);
source2 ← FloatValue64(DRh);
IF (FpuIsDisabled(sr))
   THROW FPUDIS;
result, fps ← FCMPGT_D(source1, source2, fps);
IF (FpuEnableV(fps) AND FpuCauseV(fps))
   THROW FPUEXC, fps;
Rd ← Register(result);
FPSCR ← Register(fps);
```

Description:

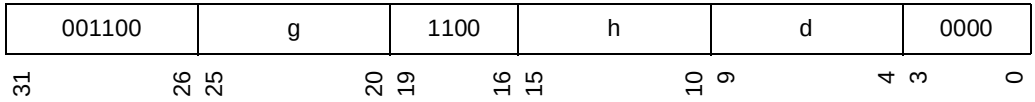
This floating-point instruction performs a double-precision floating-point greater-than comparison. It sets R_d to 1 if DR_g is greater than DR_h, and otherwise sets R_d to 0.

Possible exceptions:

FPUDIS, FPUEXC

FCMPGT.S FRg, FRh, Rd

FCMPGT.S FRg, FRh, Rd




```
sr ← ZeroExtend64(SR);
fps ← ZeroExtend32(FPSCR);
source1 ← FloatValue32(FRg);
source2 ← FloatValue32(FRh);
IF (FpuIsDisabled(sr))
   THROW FPUDIS;
result, fps ← FCMPGT_S(source1, source2, fps);
IF (FpuEnableV(fps) AND FpuCauseV(fps))
   THROW FPUEXC, fps;
Rd ← Register(result);
FPSCR ← Register(fps);
```

Description:

This floating-point instruction performs a single-precision floating-point greater-than comparison. It sets R_d to 1 if FR_g is greater than FR_h, and otherwise sets R_d to 0.

Possible exceptions:

FPUDIS, FPUEXC

FCMPGT.S and FCMPGT.D special cases:

When FPSCR.DN is 1, a positive denormalized number is treated as +0 and a negative denormalized number as -0. This flush-to-zero treatment is applied before exception detection and special case handling.

Exceptional conditions are checked in the order given below. Execution of the instruction is terminated once any check detects an exceptional condition.

1 Disabled: an exception is raised if the FPU is disabled.

2 Invalid: an invalid operation is signaled if either input is a NaN.

If the instruction does not raise an exception, a result is generated according to the following table.

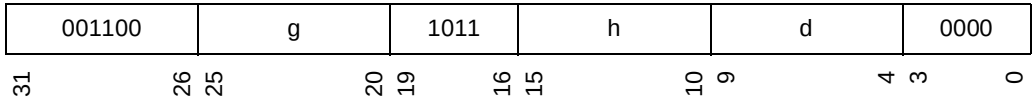
source1 → ↓ source2	+NORM, -NORM	+0	-0	+INF	-INF	+DNRM, -DNRM	qNaN	sNaN
+,-NORM	CMPGT	CMPGT	CMPGT	true	false	CMPGT	false	false
+0	CMPGT	false	false	true	false	CMPGT	false	false
-0	CMPGT	true	false	true	false	CMPGT	false	false
+INF	false	false	false	false	false	false	false	false
-INF	true	true	true	true	false	true	false	false
+, -DNRM	CMPGT	CMPGT	CMPGT	true	false	CMPGT	false	false
qNaN	false	false	false	false	false	false	false	false
sNaN	false	false	false	false	false	false	false	false

Invalid operations are indicated by light shading and raise an exception if enabled. FPU disabled cases are not shown.

The behavior of the normal 'CMPGT' case is described by the IEEE754 specification.

FCMPUN.D DRg, DRh, Rd

FCMPUN.D DRg, DRh, Rd




```
sr ← ZeroExtend64(SR);
fps ← ZeroExtend32(FPSCR);
source1 ← FloatValue64(DRg);
source2 ← FloatValue64(DRh);
IF (FpuIsDisabled(sr))
   THROW FPUDIS;
result, fps ← FCMPUN_D(source1, source2, fps);
IF (FpuEnableV(fps) AND FpuCauseV(fps))
   THROW FPUEXC, fps;
Rd ← Register(result);
FPSCR ← Register(fps);
```

Description:

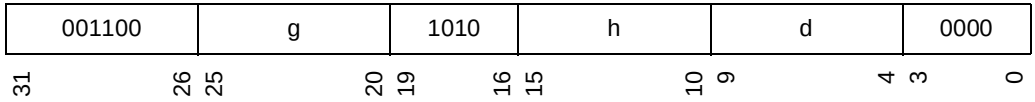
This floating-point instruction performs a double-precision floating-point unordered comparison. It sets R_d to 1 if DR_g is unordered with respect to DR_h, and otherwise sets R_d to 0.

Possible exceptions:

FPUDIS, FPUEXC

FCMPUN.S FRg, FRh, Rd

FCMPUN.S FRg, FRh, Rd




```
sr ← ZeroExtend64(SR);
fps ← ZeroExtend32(FPSCR);
source1 ← FloatValue32(FRg);
source2 ← FloatValue32(FRh);
IF (FpuIsDisabled(sr))
   THROW FPUDIS;
result, fps ← FCMPUN_S(source1, source2, fps);
IF (FpuEnableV(fps) AND FpuCauseV(fps))
   THROW FPUEXC, fps;
Rd ← Register(result);
FPSCR ← Register(fps);
```

Description:

This floating-point instruction performs a single-precision floating-point unordered comparison. It sets R_d to 1 if FR_g is unordered with respect to FR_h, and otherwise sets R_d to 0.

Possible exceptions:

FPUDIS, FPUEXC

FCMPUN.S and FCMPUN.D special cases:

When FPSCR.DN is 1, a positive denormalized number is treated as +0 and a negative denormalized number as -0. This flush-to-zero treatment is applied before exception detection and special case handling.

Exceptional conditions are checked in the order given below. Execution of the instruction is terminated once any check detects an exceptional condition.

1 Disabled: an exception is raised if the FPU is disabled.

2 Invalid: an invalid operation is signaled if either input is a signaling NaN.

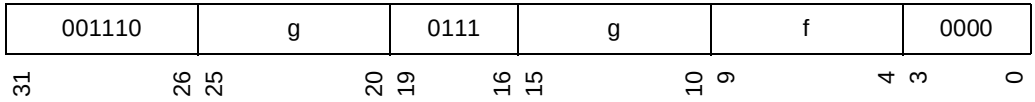
If the instruction does not raise an exception, a result is generated according to the following table.

source1 → ↓ source2	+NORM, -NORM	+0	-0	+INF	-INF	+DNRM, -DNRM	qNaN	sNaN
+,-NORM	false	false	false	false	false	false	true	true
+0	false	false	false	false	false	false	true	true
-0	false	false	false	false	false	false	true	true
+INF	false	false	false	false	false	false	true	true
-INF	false	false	false	false	false	false	true	true
+, -DNRM	false	false	false	false	false	false	true	true
qNaN	true	true	true	true	true	true	true	true
sNaN	true	true	true	true	true	true	true	true

Invalid operations are indicated by light shading and raise an exception if enabled. FPU disabled cases are not shown.

FCNV.DS DRg, FRf

FCNV.DS DRg, FRf




```
sr ← ZeroExtend32(SR);
fps ← ZeroExtend32(FPSCR);
source ← FloatValue64(DRg);
IF (FpuIsDisabled(sr))
   THROW FPUDIS;
result, fps ← FCNV_DS(source, fps);
IF (FpuEnableV(fps) AND FpuCauseV(fps))
   THROW FPUEXC, fps;
IF (FpuCauseE(fps))
   THROW FPUEXC, fps;
IF ((FpuEnableI(fps) OR FpuEnableO(fps)) OR FpuEnableU(fps))
   THROW FPUEXC, fps;
FRf ← FloatRegister32(result);
FPSCR ← Register(fps);
```

Description:

This floating-point instruction performs a double-precision to single-precision floating-point conversion. It reads a double-precision value from DR_g, converts it to single-precision and places the result in FR_f. The rounding mode is determined by FPSCR.RM.

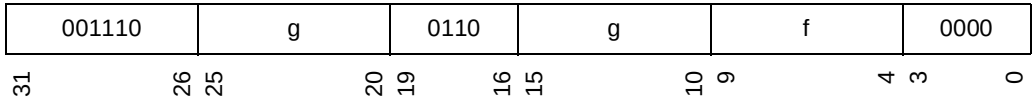
Both source operand fields must be encoded with the source register designator g.

Possible exceptions:

FPUDIS, FPUEXC

FCNV.SD FRg, DRf

FCNV.SD FRg, DRf




```
sr ← ZeroExtend64(SR);
fps ← ZeroExtend32(FPSCR);
source ← FloatValue32(FRg);
IF (FpuIsDisabled(sr))
   THROW FPUDIS;
result, fps ← FCNV_SD(source, fps);
IF (FpuEnableV(fps) AND FpuCauseV(fps))
   THROW FPUEXC, fps;
IF (FpuCauseE(fps))
   THROW FPUEXC, fps;
DRf ← FloatRegister64(result);
FPSCR ← Register(fps);
```

Description:

This floating-point instruction performs a single-precision to double-precision floating-point conversion. It reads a single-precision value from FR_g, converts it to double-precision and places the result in DR_f. FPSCR.RM has no effect since the conversion is exact.

Both source operand fields must be encoded with the source register designator g.

Possible exceptions:

FPUDIS, FPUEXC

FCNV.SD and FCNV.DS special cases:

When FPSCR.DN is 1, a positive denormalized number is treated as +0 and a negative denormalized number as -0. This flush-to-zero treatment is applied before exception detection and special case handling.

Exceptional conditions are checked in the order given below. Execution of the instruction is terminated once any check detects an exceptional condition.

1 Disabled: an exception is raised if the FPU is disabled.

2 Invalid: an invalid operation is signaled if the input is a signaling NaN.

3 Error: an FPU error is signaled if FPSCR.DN is 0 and the input is a denormalized number.

4 Inexact, underflow and overflow: these are checked together and can be signaled in combination. These cases occur for FCNV.DS but not for FCNV.SD. When inexact, underflow or overflow exceptions are requested by the user, an exception is always raised for FCNV.DS regardless of whether that condition arose.

If the instruction does not raise an exception, a result is generated according to the following table.

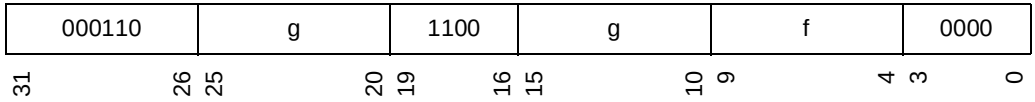
source1 →	+NORM, -NORM	+0	-0	+INF	-INF	+DNRM, -DNRM	qNaN	sNaN
CNV	+0	-0	+INF	-INF	n/a	qNaN	qNaN

FPU error is indicated by heavy shading and always raises an exception. Invalid operations are indicated by light shading and raise an exception if enabled. FPU disabled, inexact, underflow and overflow cases are not shown.

The behavior of the normal ‘CNV’ case is described by the IEEE754 specification.

FCOSA.S FRg, FRf

FCOSA.S FRg, FRf




```
sr ← ZeroExtend32(SR);
fps ← ZeroExtend32(FPSCR);
source ← FloatValue32(FRg);
IF (FpuIsDisabled(sr))
   THROW FPUDIS;
result, fps ← FCOSA_S(source, fps);
IF (FpuEnableI(fps))
   THROW FPUEXC, fps;
FRf ← FloatRegister32(result);
FPSCR ← Register(fps);
```

Description:

This floating-point instruction computes the cosine of an angle held in FR_g and places the result in FR_f. The input angle is the amount of rotation expressed as a signed fixed-point number in a 2's complement representation. The value 1 represents an angle of $360^o/2^{16}$. The upper 16 bits indicate the number of full rotations and the lower 16 bits indicate the remainder angle between 0^o and 360^o. The result is the cosine of the angle in single-precision floating-point format.

This is an approximate computation. The specified error in the result value is:

$$spec\_error = 2^{-21}.$$

Both source operands must be encoded with the source register designator g.

Possible exceptions:

FPUDIS, FPUEXC

FCOSA.S special cases:

Exceptional conditions are checked in the order given below. Execution of the instruction is terminated once any check detects an exceptional condition.

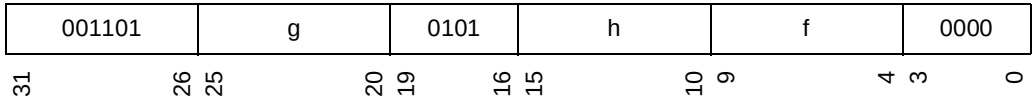
1 Disabled: an exception is raised if the FPU is disabled.

2 Inexact: this is an approximate instruction and inexact is always signaled. When inexact exceptions are requested by the user, an exception is always raised regardless of whether that condition arose. Overflow and underflow do not occur.

If the instruction does not raise an exception, the instruction computes an approximate result using an implementation-dependent algorithm.

FDIV.D DRg, DRh, DRf

FDIV.D DRg, DRh, DRf




```
sr ← ZeroExtend64(SR);
fps ← ZeroExtend32(FPSCR);
source1 ← FloatValue64(DRg);
source2 ← FloatValue64(DRh);
IF (FpuIsDisabled(sr))
   THROW FPUDIS;
result, fps ← FDIV_D(source1, source2, fps);
IF (FpuEnableV(fps) AND FpuCauseV(fps))
   THROW FPUEXC, fps;
IF (FpuEnableZ(fps) AND FpuCauseZ(fps))
   THROW FPUEXC, fps;
IF (FpuCauseE(fps))
   THROW FPUEXC, fps;
IF ((FpuEnableI(fps) OR FpuEnableO(fps)) OR FpuEnableU(fps))
   THROW FPUEXC, fps;
DRf ← FloatRegister64(result);
FPSCR ← Register(fps);
```

Description:

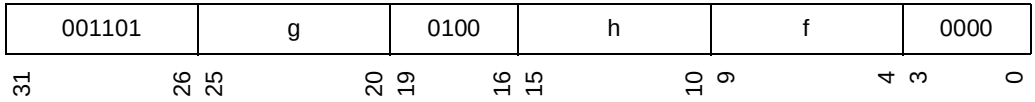
This floating-point instruction performs a double-precision floating-point division. It divides DR_g by DR_h and places the result in DR_f. The rounding mode is determined by FPSCR.RM.

Possible exceptions:

FPUDIS, FPUEXC

FDIV.S FRg, FRh, FRf

FDIV.S FRg, FRh, FRf




```
sr ← ZeroExtend64(SR);
fps ← ZeroExtend32(FPSCR);
source1 ← FloatValue32(FRg);
source2 ← FloatValue32(FRh);
IF (FpuIsDisabled(sr))
   THROW FPUDIS;
result, fps ← FDIV_S(source1, source2, fps);
IF (FpuEnableV(fps) AND FpuCauseV(fps))
   THROW FPUEXC, fps;
IF (FpuEnableZ(fps) AND FpuCauseZ(fps))
   THROW FPUEXC, fps;
IF (FpuCauseE(fps))
   THROW FPUEXC, fps;
IF ((FpuEnableI(fps) OR FpuEnableO(fps)) OR FpuEnableU(fps))
   THROW FPUEXC, fps;
FRf ← FloatRegister32(result);
FPSCR ← Register(fps);
```

Description:

This floating-point instruction performs a single-precision floating-point division. It divides FR_g by FR_h and places the result in FR_f. The rounding mode is determined by FPSCR.RM.

Possible exceptions:

FPUDIS, FPUEXC

FDIV.S and FDIV.D special cases:

When FPSCR.DN is 1, a positive denormalized number is treated as +0 and a negative denormalized number as -0. This flush-to-zero treatment is applied before exception detection and special case handling.

Exceptional conditions are checked in the order given below. Execution of the instruction is terminated once any check detects an exceptional condition.

1 Disabled: an exception is raised if the FPU is disabled.

2 Invalid: an invalid operation is signaled if either input is a signaling NaN, or if the division is of a zero by a zero, or of an infinity by an infinity.

3 Divide-by-zero: a divide-by-zero is signaled if the divisor is zero and the dividend is a finite non-zero number.

4 Error: an FPU error is signaled if FPSCR.DN is 0, neither input is a NaN and either of the following conditions is true: the divisor is a denormalized number, or the dividend is a denormalized number and the divisor is not a zero.

5 Inexact, underflow and overflow: these are checked together and can be signaled in combination. When inexact, underflow or overflow exceptions are requested by the user, an exception is always raised regardless of whether that condition arose.

If the instruction does not raise an exception, a result is generated as follows:

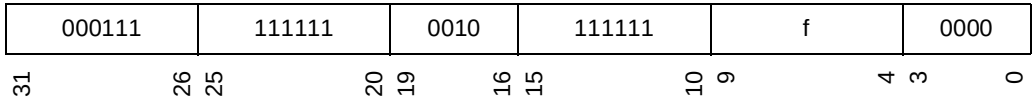
source1 → ↓ source2	+NORM, -NORM	+0	-0	+INF	-INF	+DNRM, -DNRM	qNaN	sNaN
+,-NORM	DIV	+0, -0	-0, +0	+INF, -INF	-INF, +INF	n/a	qNaN	qNaN
+0	+INF, -INF	qNaN	qNaN	+INF	-INF	+INF, -INF	qNaN	qNaN
-0	-INF, +INF	qNaN	qNaN	-INF	+INF	-INF, +INF	qNaN	qNaN
+INF	+0, -0	+0	-0	qNaN	qNaN	n/a	qNaN	qNaN
-INF	-0, +0	-0	+0	qNaN	qNaN	n/a	qNaN	qNaN
+, -DNRM	n/a	n/a	n/a	n/a	n/a	n/a	qNaN	qNaN
qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN
sNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN

FPU error is indicated by heavy shading and always raises an exception. Invalid operations and divide-by-zero are indicated by light shading and raise an exception if enabled. FPU disabled, inexact, underflow and overflow cases are not shown.

The behavior of the normal 'DIV' case is described by the IEEE754 specification.

FGETSCR FRf

FGETSCR FRf




```
sr ← ZeroExtend64(SR);
fps ← ZeroExtend32(FPSCR);
IF (FpuIsDisabled(sr))
   THROW FPUDIS;
result ← fps;
FRf ← FloatRegister32(result);
```

Description:

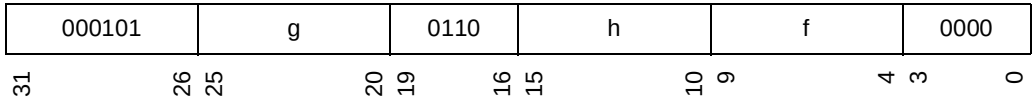
This floating-point instruction copies FPSCR to FR_f.

Possible exceptions:

FPUDIS

FIPR.S FVg, FVh, FRf

FIPR.S FVg, FVh, FVf




```
sr ← ZeroExtend64(SR);
fps ← ZeroExtend32(FPSCR);
source1 ← FloatValueVector32(FVg);
source2 ← FloatValueVector32(FVh);
IF (FpuIsDisabled(sr))
   THROW FPUDIS;
result, fps ← FIPR_S(source1, source2, fps);
IF (FpuEnableV(fps) AND FpuCauseV(fps))
   THROW FPUEXC, fps;
IF ((FpuEnableI(fps) OR FpuEnableO(fps)) OR FpuEnableU(fps))
   THROW FPUEXC, fps;
FRf ← FloatRegister32(result);
FPSCR ← Register(fps);
```

Description:

This floating-point instruction computes the dot-product of two vectors, FV_g and FV_h, and places the result in FR_f. Each vector contains four single-precision floating-point values. The dot-product is specified as:

$$FR_f = \sum_{i=0}^{3} FR_{g+i} \times FR_{h+i}$$

This is an approximate computation. The specified error in the result value is defined in *Volume 1, Chapter 8: SHmedia floating-point*.

Possible exceptions:

FPUDIS, FPUEXC

FIPR.S special cases:

FIPR.S is an approximate instruction. Denormalized numbers are supported:

- When FPSCR.DN is 0, denormalized numbers are treated as their denormalized value in the FIPR.S calculation. This instruction never signals an FPU error.
- When FPSCR.DN is 1, a positive denormalized number is treated as +0 and a negative denormalized number as -0. This flush-to-zero treatment is applied before exception detection and special case handling.

Exceptional conditions are checked in the order given below. Execution of the instruction is terminated once any check detects an exceptional condition.

1 Disabled: an exception is raised if the FPU is disabled.

2 Invalid: an invalid operation is signaled if any of the following arise:

   - Any of the inputs is a signaling NaN.
   - Multiplication of a zero by an infinity.
   - Addition of differently signed infinities where none of the inputs is a qNaN.

   The multiplication is performed with sufficient precision to avoid overflow, and therefore the multiplication of any two finite numbers does not produce an infinity. The multiplication result will be an infinity only if there is a multiplication of an infinity with a normalized number, an infinity with a denormalized number or an infinity with an infinity.

   The addition of differently signed infinities is detected if there is (at least) one positive infinity and (at least) one negative infinity in the set of 4 multiplication results.

3 Inexact, underflow and overflow: these are checked together and can be signaled in combination. This is an approximate instruction and inexact is signaled except where special cases occur. Precise details of the approximate inner-product algorithm, including the detection of underflow and overflow cases, are implementation dependent. When inexact, underflow or overflow exceptions are requested by the user, an exception is always raised regardless of whether that condition arose.

If the instruction does not raise an exception, a result is generated according to the following tables. Where the behavior is not a special case, the instruction computes an approximate result using an implementation-dependent algorithm. In the following tables, invalid operations are indicated by light shading and raise an exception if enabled. FPU disabled, inexact, underflow and overflow cases are not shown. Inexact is signaled in the 'FIPRADD' case.

Each of the 4 pairs of multiplication operands (source1 and source2) is selected from corresponding elements of the two 4-element source vectors and multiplied:

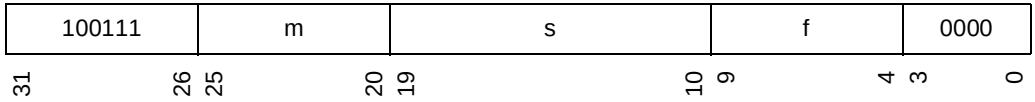
source1 → ↓ source2	+,-NORM, +,-DNRM	+0	-0	+INF	-INF	qNaN	sNaN
+,-NORM and +,-DNRM	FIPRMUL	+0, -0	-0, +0	+INF, -INF	-INF, +INF	qNaN	qNaN
+0	+0, -0	+0	-0	qNaN	qNaN	qNaN	qNaN
-0	-0, +0	-0	+0	qNaN	qNaN	qNaN	qNaN
+INF	+INF, -INF	qNaN	qNaN	+INF	-INF	qNaN	qNaN
-INF	-INF, +INF	qNaN	qNaN	-INF	+INF	qNaN	qNaN
qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN
sNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN

If any of the multiplications evaluates to qNaN, then the result of the instruction is qNaN and no further analysis need be performed. In the 'FIPRMUL', +0, -0, +INF and -INF cases, the 4 addition operands (labeled temp0 to temp3) are summed:

	temp0 →	FIPRMUL, +0, -0			+INF			-INF		
↓ temp2	temp1→ ↓ temp3	FIPRMUL, +0, -0	+INF	-INF	FIPRMUL, +0, -0	+INF	-INF	FIPRMUL, +0, -0	+INF	-INF
FIPRMUL, +0, -0	FIPRMUL, +0, -0	FIPRADD	+INF	-INF	+INF	+INF	qNaN	-INF	qNaN	-INF
	+INF	+INF	+INF	qNaN	+INF	+INF	qNaN	qNaN	qNaN	qNaN
	-INF	-INF	qNaN	-INF	qNaN	qNaN	qNaN	-INF	qNaN	-INF
+INF	FIPRMUL, +0, -0	+INF	+INF	qNaN	+INF	+INF	qNaN	qNaN	qNaN	qNaN
	+INF	+INF	+INF	qNaN	+INF	+INF	qNaN	qNaN	qNaN	qNaN
	-INF	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN
-INF	FIPRMUL, +0, -0	-INF	qNaN	-INF	qNaN	qNaN	qNaN	-INF	qNaN	-INF
	+INF	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN
	-INF	-INF	qNaN	-INF	qNaN	qNaN	qNaN	-INF	qNaN	-INF

FLD.D Rm, disp, DRf

FLD.D Rm, disp, DRf




```
sr ← ZeroExtend64(SR);
base ← ZeroExtend64(Rm);
disp ← SignExtend10(s) << 3;
IF (FpuIsDisabled(sr))
   THROW FPUDIS;
address ← ZeroExtend64(base + disp);
result ← FloatValue64(ReadMemory64(address));
DRf ← FloatRegister64(result);
```

Description:

This floating-point instruction loads a double-precision floating-point register from memory using register plus scaled immediate addressing. The effective address is formed by multiplying the sign-extended 10-bit immediate s by 8, and adding it to R_m. The 64 bits read from this effective address are loaded into DR_f.

This instruction is not considered an arithmetic operation, and it does not signal invalid operations. This instruction places no interpretation on the value transferred.

Possible exceptions:

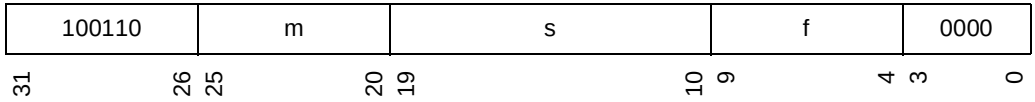
FPUDIS, RADDERR, RTLBMISS, READPROT

Notes:

The ‘disp’ in the assembly syntax represents the immediate s after sign extension and scaling.

FLD.P Rm, disp, FPf

FLD.P Rm, disp, FPf




```
sr ← ZeroExtend64(SR);
base ← ZeroExtend64(Rm);
disp ← SignExtend10(s) << 3;
IF (FpuIsDisabled(sr))
   THROW FPUDIS;
address ← ZeroExtend64(base + disp);
result ← ReadMemoryPair32(address);
FPf ← FloatRegisterPair32(result);
```

Description:

This floating-point instruction loads a pair of single-precision floating-point registers from memory using register plus scaled immediate addressing. The effective address (EA) is formed by multiplying the sign-extended 10-bit immediate s by 8, and adding it to R_m. The 64 bits of data read from this effective address are loaded into FP_f as a pair of single-precision floating-point values. The 32 bits of data read from EA are placed in FR_f, and the 32 bits of data read from EA+4 are placed in FR_{f+1}.

This instruction is not considered an arithmetic operation, and it does not signal invalid operations. This instruction places no interpretation on the value transferred.

Possible exceptions:

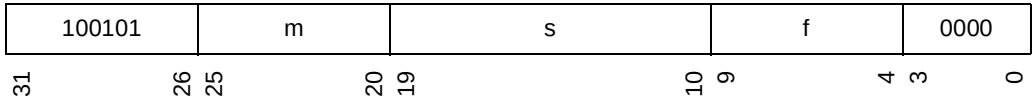
FPUDIS, RADDERR, RTLBMISS, READPROT

Notes:

The ‘disp’ in the assembly syntax represents the immediate s after sign extension and scaling. The memory representation of pairs of single-precision floating-point registers is defined in *Volume 1, Chapter 3: Data representation*.

FLD.S Rm, disp, FRf

FLD.S Rm, disp, FRf




```
sr ← ZeroExtend64(SR);
base ← ZeroExtend64(Rm);
disp ← SignExtend10(s) << 2;
IF (FpuIsDisabled(sr))
   THROW FPUDIS;
address ← ZeroExtend64(base + disp);
result ← FloatValue32(ReadMemory32(address));
FRf ← FloatRegister32(result);
```

Description:

This floating-point instruction loads a single-precision floating-point register from memory using register plus scaled immediate addressing. The effective address is formed by multiplying the sign-extended 10-bit immediate s by 4, and adding it to R_m. The 32 bits read from this effective address are loaded into FR_f.

This instruction is not considered an arithmetic operation, and it does not signal invalid operations. This instruction places no interpretation on the value transferred.

Possible exceptions:

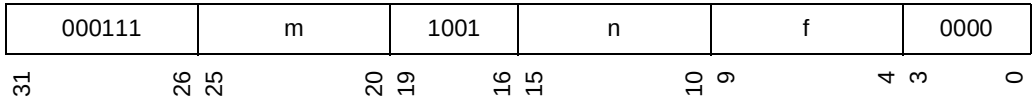
FPUDIS, RADDERR, RTLBMISS, READPROT

Notes:

The ‘disp’ in the assembly syntax represents the immediate s after sign extension and scaling.

FLDX.D Rm, Rn, DRf

FLDX.D Rm, Rn, DRf




```
sr ← ZeroExtend64(SR);
base ← ZeroExtend64(Rm);
index ← SignExtend64(Rn);
IF (FpuIsDisabled(sr))
   THROW FPUDIS;
address ← ZeroExtend64(base + index);
result ← FloatValue64(ReadMemory64(address));
DRf ← FloatRegister64(result);
```

Description:

This floating-point instruction loads a double-precision floating-point register from memory using register plus register addressing. The effective address is formed by adding R_m to R_n. The 64 bits read from this effective address are loaded into DR_f.

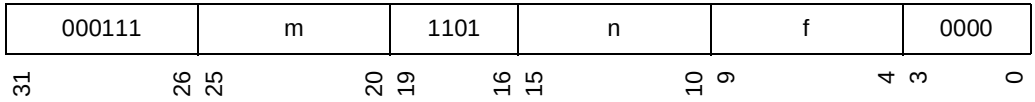
This instruction is not considered an arithmetic operation, and it does not signal invalid operations. This instruction places no interpretation on the value transferred.

Possible exceptions:

FPUDIS, RADDERR, RTLBMISS, READPROT

FLDX.P Rm, Rn, FPf

FLDX.P Rm, Rn, FPf




```
sr ← ZeroExtend64(SR);
base ← ZeroExtend64(Rm);
index ← SignExtend64(Rn);
IF (FpuIsDisabled(sr))
   THROW FPUDIS;
address ← ZeroExtend64(base + index);
result ← ReadMemoryPair32(address);
FPf ← FloatRegisterPair32(result);
```

Description:

This floating-point instruction loads a pair of single-precision floating-point registers from memory using register plus register addressing. The effective address (EA) is formed by adding R_m to R_n. The 64 bits of data read from this effective address are loaded into FP_f as a pair of single-precision floating-point values. The 32 bits of data read from EA are placed in FR_f, and the 32 bits of data read from EA+4 are placed in FR_{f+1}.

This instruction is not considered an arithmetic operation, and it does not signal invalid operations. This instruction places no interpretation on the value transferred.

Possible exceptions:

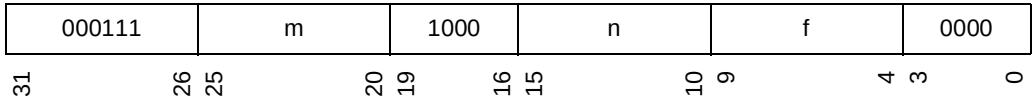
FPUDIS, RADDERR, RTLBMISS, READPROT

Notes:

The memory representation of pairs of single-precision floating-point registers is defined in *Volume 1, Chapter 3: Data representation*.

FLDX.S Rm, Rn, FRf

FLDX.S Rm, Rn, FRf




```
sr ← ZeroExtend64(SR);
base ← ZeroExtend64(Rm);
index ← SignExtend64(Rn);
IF (FpuIsDisabled(sr))
   THROW FPUDIS;
address ← ZeroExtend64(base + index);
result ← FloatValue32(ReadMemory32(address));
FRf ← FloatRegister32(result);
```

Description:

This floating-point instruction loads a single-precision floating-point register from memory using register plus register addressing. The effective address is formed by adding R_m to R_n. The 32 bits read from this effective address are loaded into FR_f.

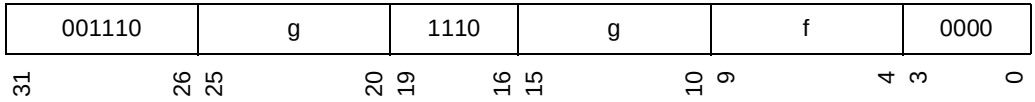
This instruction is not considered an arithmetic operation, and it does not signal invalid operations. This instruction places no interpretation on the value transferred.

Possible exceptions:

FPUDIS, RADDERR, RTLBMISS, READPROT

FLOAT.LD FRg, DRf

FLOAT.LD FRg, DRf




```
sr ← ZeroExtend32(SR);
fps ← ZeroExtend32(FPSCR);
source ← FloatValue32(FRg);
IF (FpuIsDisabled(sr))
   THROW FPUDIS;
result, fps ← FLOAT_LD(source, fps);
DRf ← FloatRegister64(result);
FPSCR ← Register(fps);
```

Description:

This floating-point instruction performs a signed 32-bit integer to double-precision floating-point conversion. It reads a signed 32-bit integer value from FR_g, converts it to a double-precision range and places the result in DR_f. In all cases the provided integer value will be exactly represented in the destination floating-point format. FPSCR.RM has no effect since the conversion is exact.

If the required source value is held in the general-purpose register file, it is necessary to move it to the floating-point register file before the conversion.

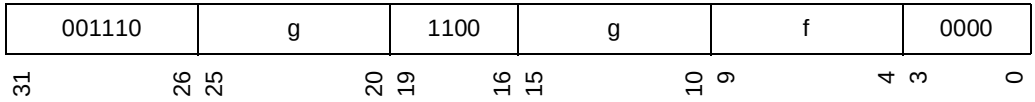
Both source operand fields must be encoded with the source register designator g.

Possible exceptions:

FPUDIS

FLOAT.LS FRg, FRf

FLOAT.LS FRg, FRf




```
sr ← ZeroExtend32(SR);
fps ← ZeroExtend32(FPSCR);
source ← FloatValue32(FRg);
IF (FpuIsDisabled(sr))
   THROW FPUDIS;
result, fps ← FLOAT_LS(source, fps);
IF (FpuEnableI(fps))
   THROW FPUEXC, fps;
FRf ← FloatRegister32(result);
FPSCR ← Register(fps);
```

Description:

This floating-point instruction performs a signed 32-bit integer to single-precision floating-point conversion. It reads a signed 32-bit integer value from FR_g, converts it to a single-precision range and places the result in FR_f. In cases where the integer value cannot be exactly represented in the destination floating-point format, the rounding mode is determined by FPSCR.RM.

If the required source value is held in the general-purpose register file, it is necessary to move it to the floating-point register file before the conversion.

Both source operand fields must be encoded with the source register designator g.

Possible exceptions:

FPUDIS, FPUEXC

FLOAT.LS and FLOAT.LD special cases:

Exceptional conditions are checked in the order given below. Execution of the instruction is terminated once any check detects an exceptional condition.

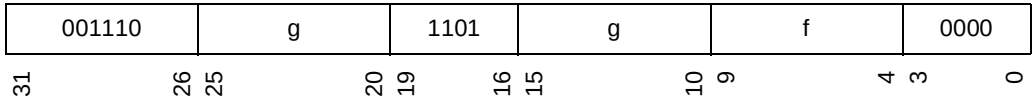
1 Disabled: an exception is raised if the FPU is disabled.

2 Inexact: inexact can occur for FLOAT.LS but not for FLOAT.LD. When inexact exceptions are requested by the user, an exception is always raised for FLOAT.LS regardless of whether that condition arose. Overflow and underflow do not occur for either of these instructions.

If the instruction does not raise an exception, the conversion is performed as indicated by the IEEE754 specification.

FLOAT.QD DRg, DRf

FLOAT.QD DRg, DRf




```
sr ← ZeroExtend32(SR);
fps ← ZeroExtend32(FPSCR);
source ← FloatValue64(DRg);
IF (FpuIsDisabled(sr))
   THROW FPUDIS;
result, fps ← FLOAT_QD(source, fps);
IF (FpuEnableI(fps))
   THROW FPUEXC, fps;
DRf ← FloatRegister64(result);
FPSCR ← Register(fps);
```

Description:

This floating-point instruction performs a signed 64-bit integer to double-precision floating-point conversion. It reads a signed 64-bit integer value from DR_g, converts it to a double-precision range and places the result in DR_f. In cases where the integer value cannot be exactly represented in the destination floating-point format, the rounding mode is determined by FPSCR.RM.

If the required source value is held in the general-purpose register file, it is necessary to move it to the floating-point register file before the conversion.

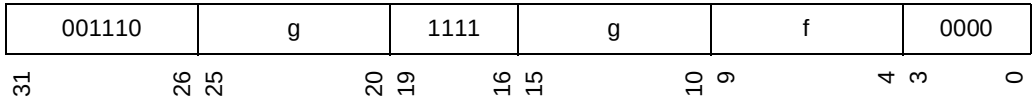
Both source operand fields must be encoded with the source register designator g.

Possible exceptions:

FPUDIS, FPUEXC

FLOAT.QS DRg, FRf

FLOAT.QS DRg, FRf




```
sr ← ZeroExtend32(SR);
fps ← ZeroExtend32(FPSCR);
source ← FloatValue64(DRg);
IF (FpuIsDisabled(sr))
   THROW FPUDIS;
result, fps ← FLOAT_QS(source, fps);
IF (FpuEnableI(fps))
   THROW FPUEXC, fps;
FRf ← FloatRegister32(result);
FPSCR ← Register(fps);
```

Description:

This floating-point instruction performs a signed 64-bit integer to single-precision floating-point conversion. It reads a signed 64-bit integer value from DR_g, converts it to a single-precision range and places the result in FR_f. In cases where the integer value cannot be exactly represented in the destination floating-point format, the rounding mode is determined by FPSCR.RM.

If the required source value is held in the general-purpose register file, it is necessary to move it to the floating-point register file before the conversion.

Both source operand fields must be encoded with the source register designator g.

Possible exceptions:

FPUDIS, FPUEXC

FLOAT.QS and FLOAT.QD special cases:

Exceptional conditions are checked in the order given below. Execution of the instruction is terminated once any check detects an exceptional condition.

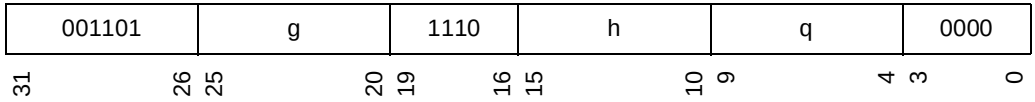
1 Disabled: an exception is raised if the FPU is disabled.

2 Inexact: inexact can occur for both of these instructions.When inexact exceptions are requested by the user, an exception is always raised regardless of whether that condition arose. Overflow and underflow do not occur.

If the instruction does not raise an exception, the conversion is performed as indicated by the IEEE754 specification.

FMAC.S FRg, FRh, FRq

FMAC.S FRg, FRh, FRq




```
sr ← ZeroExtend64(SR);
fps ← ZeroExtend32(FPSCR);
source1 ← FloatValue32(FRg);
source2 ← FloatValue32(FRh);
source3_result ← FloatValue32(FRq);
IF (FpuIsDisabled(sr))
   THROW FPUDIS;
source3_result, fps ← FMAC_S(source1, source2, source3_result, fps);
IF (FpuEnableV(fps) AND FpuCauseV(fps))
   THROW FPUEXC, fps;
IF (FpuCauseE(fps))
   THROW FPUEXC, fps;
IF ((FpuEnableI(fps) OR FpuEnableO(fps)) OR FpuEnableU(fps))
   THROW FPUEXC, fps;
FRq ← FloatRegister32(source3_result);
FPSCR ← Register(fps);
```

Description:

This floating-point instruction performs a single-precision floating-point multiply-accumulate. It multiplies FR_g by FR_h, adds this intermediate to FR_q and places the result in FR_q.

The multiplication and addition are performed as if the exponent and precision ranges were unbounded, followed by one rounding down to single-precision format. The rounding mode is determined by FPSCR.RM.

Possible exceptions:

FPUDIS, FPUEXC

FMAC.S special cases:

When FPSCR.DN is 1, a positive denormalized number is treated as +0 and a negative denormalized number as -0. This flush-to-zero treatment is applied before exception detection and special case handling.

Exceptional conditions are checked in the order given below. Execution of the instruction is terminated once any check detects an exceptional condition.

1 Disabled: an exception is raised if the FPU is disabled.

2 Invalid: an invalid operation is signaled if any of the three inputs is a signaling NaN, there is a multiplication of a zero by an infinity, or there is an addition of differently signed infinities.

  The multiplication is performed with sufficient precision to avoid overflow, and therefore the multiplication of any two finite numbers does not produce an infinity. The multiplication result will be an infinity only if there is a multiplication of an infinity with a normalized number, an infinity with a denormalized number or an infinity with an infinity.

3 Error: an FPU error is signaled if FPSCR.DN is 0 and none of the inputs are a NaN and at least one of the inputs is a denormalized number.

4 Inexact, underflow and overflow: these are checked together and can be signaled in combination. The multiply-accumulate is implemented using a fused-mac algorithm, and these are detected during the conversion of the exactly evaluated intermediate to the single-precision result. When inexact, underflow or overflow exceptions are requested by the user, an exception is always raised regardless of whether that condition arose.

If the instruction does not raise an exception, a result is generated according to the following tables. In these tables, FPU error is indicated by heavy shading and always raises an exception. Invalid operations are indicated by light shading and raise an exception if enabled. FPU disabled, inexact, underflow and overflow cases are not shown.

Firstly, the operands are checked for sNaN:

source1 →	other		sNaN	
source2 → ↓ source3_result	other	sNaN	other	sNaN
other		qNaN	qNaN	qNaN
sNaN	qNaN	qNaN	qNaN	qNaN

If the result of the previous table is a qNaN, no further analysis is performed. In all other cases, source1 and source2 are checked for a zero multiplied by an infinity:

↓ source2, source1 →	other	+0	-0	+INF	-INF
other					
+0				qNaN	qNaN
-0				qNaN	qNaN
+INF		qNaN	qNaN		
-INF		qNaN	qNaN		

If the result of the previous table is a qNaN, no further analysis is performed. In all other cases, the operands are checked for input qNaN values:

source1 →	other		qNaN	
↓ source3_result, source2 →	other	qNaN	other	qNaN
other		qNaN	qNaN	qNaN
qNaN	qNaN	qNaN	qNaN	qNaN

By this stage all operations involving sNaN or qNaN operands have been dealt with. If the result of the previous table is a qNaN, no further analysis is performed. In all other cases, the operands are checked for the addition of differently signed infinities:

source1 →	+other				-other				+INF				-INF			
source2 → ↓ source3_result	+other	-other	+INF	-INF	+other	-other	+INF	-INF	+other	-other	+INF	-INF	+other	-other	+INF	-INF
+other, -other																
+INF				qNaN			qNaN			qNaN		qNaN	qNaN		qNaN	
-INF			qNaN				qNaN		qNaN		qNaN			qNaN		qNaN

If the result of the previous table is a qNaN, no further analysis is performed. In all other cases, source1 and source2 are multiplied:

source1 → ↓ source2	+NORM, -NORM	+0	-0	+INF	-INF	+DNRM, -DNRM
+,-NORM	FULLMUL	+0, -0	-0, +0	+INF, -INF	-INF, +INF	n/a
+0	+0, -0	+0	-0			n/a
-0	-0, +0	-0	+0			n/a
+INF	+INF, -INF			+INF	-INF	n/a
-INF	-INF, +INF			-INF	+INF	n/a
+, -DNRM	n/a	n/a	n/a	n/a	n/a	n/a

The empty cells in this table correspond to cases that have already been dealt with. If either source is denormalized, no further analysis is performed. In the 'FULLMUL' case, a multiplication is performed without loss of precision. There is no rounding nor overflow, and this multiplication cannot produce an intermediate infinity.

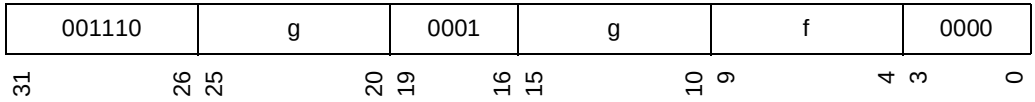
In the 'FULLMUL', +0, -0, +INF and -INF cases, the 2 addition operands (source1*source2 and source3_result) are summed:

(source1*source2)→ ↓ source3_result	FULLMUL	+0	-0	+INF	-INF
+,-NORM	FULLADD	source3_result	source3_result	+INF	-INF
+0	FULLADD	+0	+0	+INF	-INF
-0	FULLADD	+0	-0	+INF	-INF
+INF	+INF	+INF	+INF	+INF	
-INF	-INF	-INF	-INF		-INF
+, -DNRM	n/a	n/a	n/a	n/a	n/a

The two empty cells in this table correspond to cases that have already been dealt with. In the 'FULLADD' cases the fully-precise addition intermediate is rounded to give a single-precision result.

FMOV.D DRg, DRf

FMOV.D DRg, DRf




```
sr ← ZeroExtend64(SR);
source ← FloatValue64(DRg);
IF (FpuIsDisabled(sr))
   THROW FPUDIS;
result ← source;
DRf ← FloatRegister64(result);
```

Description:

This floating-point instruction reads a double-precision floating-point value from DR_g and copies it to DR_f. This is a bit-by-bit copy with no interpretation or conversion of the value.

This instruction is not considered an arithmetic operation, and it does not signal invalid operations. There are no special floating-point cases associated with this instruction.

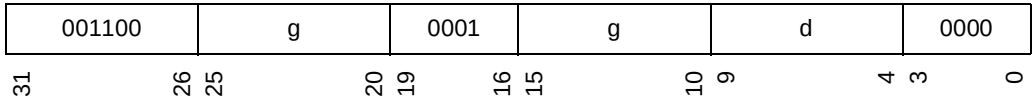
Both source operand fields must be encoded with the source register designator g.

Possible exceptions:

FPUDIS

FMOV.DQ DRg, Rd

FMOV.DQ DRg, Rd




```
sr ← ZeroExtend64(SR);
source ← FloatValue64(DRg);
IF (FpuIsDisabled(sr))
   THROW FPUDIS;
result ← source;
Rd ← Register(result);
```

Description:

This floating-point instruction reads a double-precision floating-point value from DR_g and copies it to R_d. This is a bit-by-bit copy with no interpretation or conversion of the value. The lower 32 bits of R_d will hold the lower 32 bits of DR_g (also known as FR_{g+1}), and the upper 32 bits of R_d will hold the upper 32 bits of DR_g (also known as FR_g).

This instruction is not considered an arithmetic operation, and it does not signal invalid operations. There are no special floating-point cases associated with this instruction.

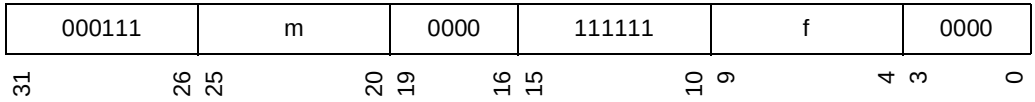
Both source operand fields must be encoded with the source register designator g.

Possible exceptions:

FPUDIS

FMOV.LS Rm, FRf

FMOV.LS Rm, FRf




```
sr ← ZeroExtend64(SR);
source ← SignExtend32(Rm);
IF (FpuIsDisabled(sr))
   THROW FPUDIS;
result ← source;
FRf ← FloatRegister32(result);
```

Description:

This floating-point instruction reads the lower 32 bits of R_d and copies that bit-value to the single-precision floating-point register FR_f. This is a bit-by-bit copy with no interpretation or conversion of the value. FR_f will hold the lower 32 bits of R_m, while the upper 32 bits of R_m are ignored.

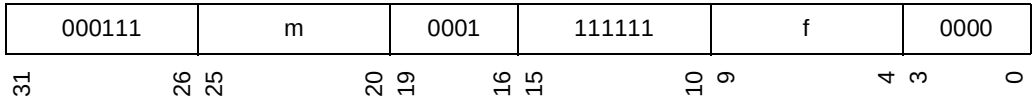
This instruction is not considered an arithmetic operation, and it does not signal invalid operations. There are no special floating-point cases associated with this instruction.

Possible exceptions:

FPUDIS

FMOV.QD Rm, DRf

FMOV.QD Rm, DRf




```
sr ← ZeroExtend64(SR);
source ← SignExtend64(Rm);
IF (FpuIsDisabled(sr))
   THROW FPUDIS;
result ← source;
DRf ← FloatRegister64(result);
```

Description:

This floating-point instruction reads all 64 bits of R_d and copies that bit-value to the double-precision floating-point register DR_f. This is a bit-by-bit copy with no interpretation or conversion of the value. The lower 32 bits of DR_f (also known as FR_{f+1}) will hold the lower 32 bits of R_m, and the upper 32 bits of DR_f (also known as FR_f) will hold the upper 32 bits of R_m.

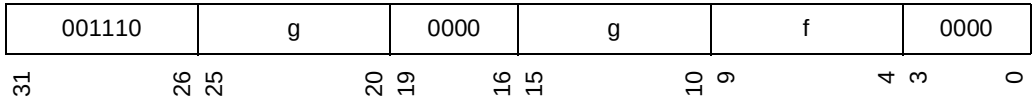
This instruction is not considered an arithmetic operation, and it does not signal invalid operations. There are no special floating-point cases associated with this instruction.

Possible exceptions:

FPUDIS

FMOV.S FRg, FRf

FMOV.S FRg, FRf




```
sr ← ZeroExtend64(SR);
source ← FloatValue32(FRg);
IF (FpuIsDisabled(sr))
   THROW FPUDIS;
result ← source;
FRf ← FloatRegister32(result);
```

Description:

This floating-point instruction reads a single-precision floating-point value from FR_g and copies it to FR_f. This is a bit-by-bit copy with no interpretation or conversion of the value.

This instruction is not considered an arithmetic operation, and it does not signal invalid operations. There are no special floating-point cases associated with this instruction.

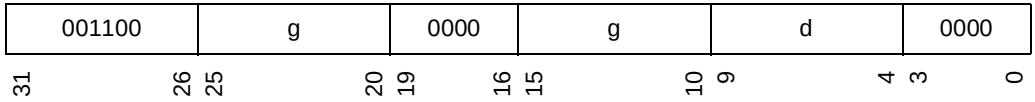
Both source operand fields must be encoded with the source register designator g.

Possible exceptions:

FPUDIS

FMOV.SL FRg, Rd

FMOV.SL FRg, Rd




```
sr ← ZeroExtend64(SR);
source ← FloatValue32(FRg);
IF (FpuIsDisabled(sr))
   THROW FPUDIS;
result ← SignExtend32(source);
Rd ← Register(result);
```

Description:

This floating-point instruction reads a single-precision floating-point value from FR_g and copies it to R_d. This is a bit-by-bit copy with no interpretation or conversion of the value. The lower 32 bits of R_d will hold the bit-value of FR_g, and the upper 32 bits of R_d will be sign extensions of R_d's $31^{st.}$ bit.

This instruction is not considered an arithmetic operation, and it does not signal invalid operations. There are no special floating-point cases associated with this instruction.

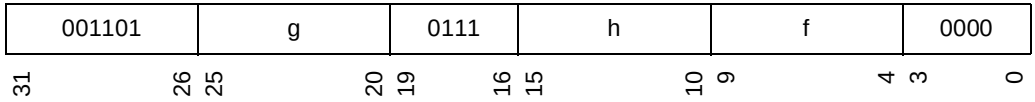
Both source operand fields must be encoded with the source register designator g.

Possible exceptions:

FPUDIS

FMUL.D DRg, DRh, DRf

FMUL.D DRg, DRh, DRf




```
sr ← ZeroExtend64(SR);
fps ← ZeroExtend32(FPSCR);
source1 ← FloatValue64(DRg);
source2 ← FloatValue64(DRh);
IF (FpuIsDisabled(sr))
   THROW FPUDIS;
result, fps ← FMUL_D(source1, source2, fps);
IF (FpuEnableV(fps) AND FpuCauseV(fps))
   THROW FPUEXC, fps;
IF (FpuCauseE(fps))
   THROW FPUEXC, fps;
IF ((FpuEnableI(fps) OR FpuEnableO(fps)) OR FpuEnableU(fps))
   THROW FPUEXC, fps;
DRf ← FloatRegister64(result);
FPSCR ← Register(fps);
```

Description:

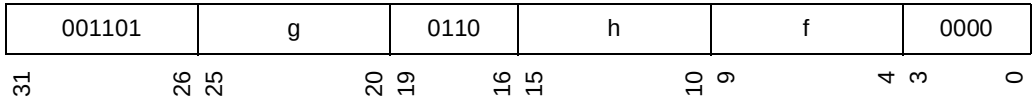
This floating-point instruction performs a double-precision floating-point multiplication. It multiplies DR_g by DR_h and places the result in DR_f. The rounding mode is determined by FPSCR.RM.

Possible exceptions:

FPUDIS, FPUEXC

FMUL.S FRg, FRh, FRf

FMUL.S FRg, FRh, FRf




```
sr ← ZeroExtend64(SR);
fps ← ZeroExtend32(FPSCR);
source1 ← FloatValue32(FRg);
source2 ← FloatValue32(FRh);
IF (FpuIsDisabled(sr))
   THROW FPUDIS;
result, fps ← FMUL_S(source1, source2, fps);
IF (FpuEnableV(fps) AND FpuCauseV(fps))
   THROW FPUEXC, fps;
IF (FpuCauseE(fps))
   THROW FPUEXC, fps;
IF ((FpuEnableI(fps) OR FpuEnableO(fps)) OR FpuEnableU(fps))
   THROW FPUEXC, fps;
FRf ← FloatRegister32(result);
FPSCR ← Register(fps);
```

Description:

This floating-point instruction performs a single-precision floating-point multiplication. It multiplies FR_g by FR_h and places the result in FR_f. The rounding mode is determined by FPSCR.RM.

Possible exceptions:

FPUDIS, FPUEXC

FMUL.S and FMUL.D special cases:

When FPSCR.DN is 1, a positive denormalized number is treated as +0 and a negative denormalized number as -0. This flush-to-zero treatment is applied before exception detection and special case handling.

Exceptional conditions are checked in the order given below. Execution of the instruction is terminated once any check detects an exceptional condition.

1 Disabled: an exception is raised if the FPU is disabled.

2 Invalid: an invalid operation is signaled if either input is a signaling NaN, or if this is a multiplication of a zero by an infinity.

3 Error: an FPU error is signaled if FPSCR.DN is 0, neither input is a NaN and either input is a denormalized number.

4 Inexact, underflow and overflow: these are checked together and can be signaled in combination. When inexact, underflow or overflow exceptions are requested by the user, an exception is always raised regardless of whether that condition arose.

If the instruction does not raise an exception, a result is generated according to the following table.

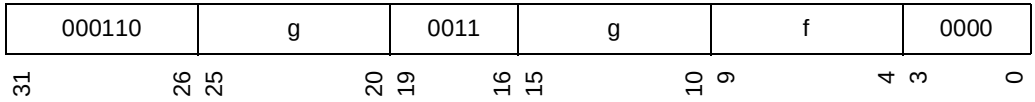
source1 → ↓ source2	+NORM, -NORM	+0	-0	+INF	-INF	+DNRM, -DNRM	qNaN	sNaN
+,-NORM	MUL	+0, -0	-0, +0	+INF, -INF	-INF, +INF	n/a	qNaN	qNaN
+0	+0, -0	+0	-0	qNaN	qNaN	n/a	qNaN	qNaN
-0	-0, +0	-0	+0	qNaN	qNaN	n/a	qNaN	qNaN
+INF	+INF, -INF	qNaN	qNaN	+INF	-INF	n/a	qNaN	qNaN
-INF	-INF, +INF	qNaN	qNaN	-INF	+INF	n/a	qNaN	qNaN
+, -DNRM	n/a	n/a	n/a	n/a	n/a	n/a	qNaN	qNaN
qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN
sNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN

FPU error is indicated by heavy shading and always raises an exception. Invalid operations are indicated by light shading and raise an exception if enabled. FPU disabled, inexact, underflow and overflow cases are not shown.

The behavior of the normal 'MUL' case is described by the IEEE754 specification.

FNEG.D DRg, DRf

FNEG.D DRg, DRf




```
sr ← ZeroExtend64(SR);
source ← FloatValue64(DRg);
IF (FpuIsDisabled(sr))
   THROW FPUDIS;
result ← FNEG_D(source);
DRf ← FloatRegister64(result);
```

Description:

This floating-point instruction computes the negated value of a double-precision floating-point number. It reads DR_g, inverts the sign bit and places the result in DR_f.

This instruction is not considered an arithmetic operation, and it does not signal invalid operations. There are no special floating-point cases associated with this instruction.

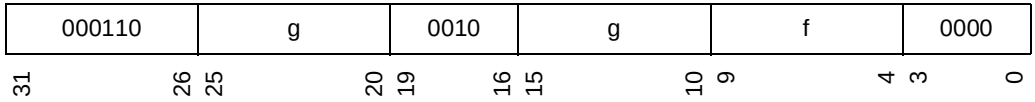
Both source operand fields must be encoded with the source register designator g.

Possible exceptions:

FPUDIS

FNEG.S FRg, FRf

FNEG.S FRg, FRf




```
sr ← ZeroExtend64(SR);
source ← FloatValue32(FRg);
IF (FpuIsDisabled(sr))
   THROW FPUDIS;
result ← FNEG_S(source);
FRf ← FloatRegister32(result);
```

Description:

This floating-point instruction computes the negated value of a single-precision floating-point number. It reads FR_g, inverts the sign bit and places the result in FR_f.

This instruction is not considered an arithmetic operation, and it does not signal invalid operations. There are no special floating-point cases associated with this instruction.

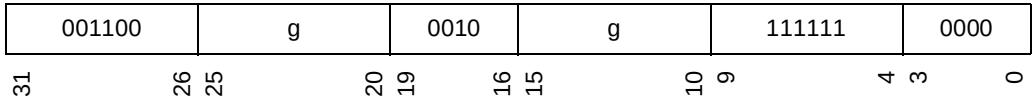
Both source operand fields must be encoded with the source register designator g.

Possible exceptions:

FPUDIS

FPUTSCR FRg

FPUTSCR FRg




```
sr ← ZeroExtend64(SR);
source ← FloatValue32(FRg);
IF (FpuIsDisabled(sr))
   THROW FPUDIS;
fps ← source;
FPSCR ← Register(fps);
```

Description:

This floating-point instruction copies FR_g to FPSCR. This setting of FPSCR does not cause any floating-point exceptional conditions to be signaled.

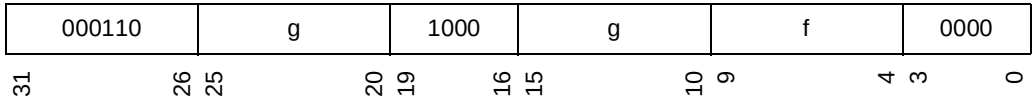
Both source operand fields must be encoded with the source register designator g.

Possible exceptions:

FPUDIS

FSINA.S FRg, FRf

FSINA.S FRg, FRf




```
sr ← ZeroExtend32(SR);
fps ← ZeroExtend32(FPSCR);
source ← FloatValue32(FRg);
IF (FpuIsDisabled(sr))
   THROW FPUDIS;
result, fps ← FSINA_S(source, fps);
IF (FpuEnableI(fps))
   THROW FPUEXC, fps;
FRf ← FloatRegister32(result);
FPSCR ← Register(fps);
```

Description:

This floating-point instruction computes the sine of an angle held in FR_g and places the result in FR_f. The input angle is the amount of rotation expressed as a signed fixed-point number in a 2's complement representation. The value 1 represents an angle of $360^o/2^{16}$. The upper 16 bits indicate the number of full rotations and the lower 16 bits indicate the remainder angle between 0^o and 360^o. The result is the sine of the angle in single-precision floating-point format.

This is an approximate computation. The specified error in the result value is:

$$spec\_error = 2^{-21}.$$

Both source operands must be encoded with the source register designator g.

Possible exceptions:

FPUDIS, FPUEXC

FSINA.S special cases:

Exceptional conditions are checked in the order given below. Execution of the instruction is terminated once any check detects an exceptional condition.

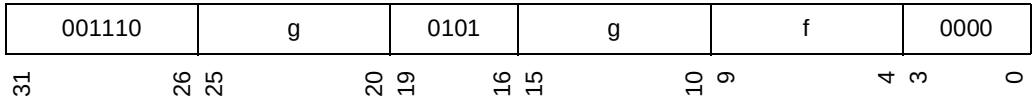
1 Disabled: an exception is raised if the FPU is disabled.

2 Inexact: this is an approximate instruction and inexact is always signaled. When inexact exceptions are requested by the user, an exception is always raised regardless of whether that condition arose. Overflow and underflow do not occur.

If the instruction does not raise an exception, the instruction computes an approximate result using an implementation-dependent algorithm.

FSQRT.D DRg, DRf

FSQRT.D DRg, DRf




```
sr ← ZeroExtend64(SR);
fps ← ZeroExtend32(FPSCR);
source ← FloatValue64(DRg);
IF (FpuIsDisabled(sr))
   THROW FPUDIS;
result, fps ← FSQRT_D(source, fps);
IF (FpuEnableV(fps) AND FpuCauseV(fps))
   THROW FPUEXC, fps;
IF (FpuCauseE(fps))
   THROW FPUEXC, fps;
IF (FpuEnableI(fps))
   THROW FPUEXC, fps;
DRf ← FloatRegister64(result);
FPSCR ← Register(fps);
```

Description:

This floating-point instruction performs a double-precision floating-point square root. It extracts the square root of DR_g and places the result in DR_f. The rounding mode is determined by FPSCR.RM.

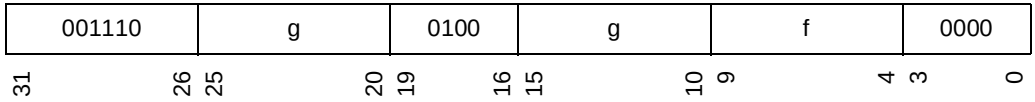
Both source operand fields must be encoded with the source register designator g.

Possible exceptions:

FPUDIS, FPUEXC

FSQRT.S FRg, FRf

FSQRT.S FRg, FRf




```
sr ← ZeroExtend64(SR);
fps ← ZeroExtend32(FPSCR);
source ← FloatValue32(FRg);
IF (FpuIsDisabled(sr))
   THROW FPUDIS;
result, fps ← FSQRT_S(source, fps);
IF (FpuEnableV(fps) AND FpuCauseV(fps))
   THROW FPUEXC, fps;
IF (FpuCauseE(fps))
   THROW FPUEXC, fps;
IF (FpuEnableI(fps))
   THROW FPUEXC, fps;
FRf ← FloatRegister32(result);
FPSCR ← Register(fps);
```

Description:

This floating-point instruction performs a single-precision floating-point square root. It extracts the square root of FR_g and places the result in FR_f. The rounding mode is determined by FPSCR.RM.

Both source operand fields must be encoded with the source register designator g.

Possible exceptions:

FPUDIS, FPUEXC

FSQRT.S and FSQRT.D special cases:

When FPSCR.DN is 1, a positive denormalized number is treated as +0 and a negative denormalized number as -0. This flush-to-zero treatment is applied before exception detection and special case handling.

Exceptional conditions are checked in the order given below. Execution of the instruction is terminated once any check detects an exceptional condition.

1 Disabled: an exception is raised if the FPU is disabled.

2 Invalid: an invalid operation is signaled if the input is a signaling NaN, or if this is a square root of a number less than zero (including negative infinity and negative normalized/denormalized numbers, but excluding negative zero).

3 Error: an FPU error is signaled if FPSCR.DN is 0 and the input is a positive denormalized number.

4 Inexact: only inexact is checked. When inexact exceptions are requested by the user, an exception is always raised regardless of whether that condition arose. Overflow and underflow do not occur.

If the instruction does not raise an exception, a result is generated according to the following table.

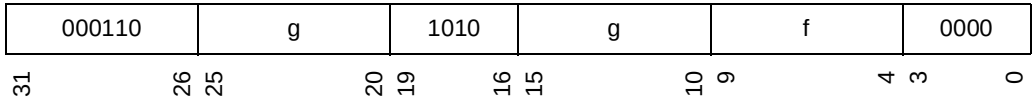
source1 →	+NORM	-NORM	+0	-0	+INF	-INF	+DNRM	-DNRM	qNaN	sNaN
SQRT	qNaN	+0	-0	+INF	qNaN	n/a	qNaN	qNaN	qNaN

FPU error is indicated by heavy shading and always raises an exception. Invalid operations are indicated by light shading and raise an exception if enabled. FPU disabled and inexact cases are not shown.

The behavior of the normal 'SQRT' case is described by the IEEE754 specification.

FSRRA.S FRg, FRf

FSRRA.S FRg, FRf




```
sr ← ZeroExtend64(SR);
fps ← ZeroExtend32(FPSCR);
source ← FloatValue32(FRg);
IF (FpuIsDisabled(sr))
   THROW FPUDIS;
result, fps ← FSRRA_S(source, fps);
IF (FpuEnableV(fps) AND FpuCauseV(fps))
   THROW FPUEXC, fps;
IF (FpuEnableZ(fps) AND FpuCauseZ(fps))
   THROW FPUEXC, fps;
IF (FpuCauseE(fps))
   THROW FPUEXC, fps;
IF (FpuEnableI(fps))
   THROW FPUEXC, fps;
FRf ← FloatRegister32(result);
FPSCR ← Register(fps);
```

Description:

This floating-point instruction computes the reciprocal of the square root of the value held in FR_g and places the result in FR_f. This is an approximate computation. The specified error in the result value is:

$$spec\_error = 2^{E-21}$$

where E = unbiased exponent value of result.

Both source operands must be encoded with the source register designator g.

Possible exceptions:

FPUDIS, FPUEXC

FSRRA.S special cases:

When FPSCR.DN is 1, a positive denormalized number is treated as +0 and a negative denormalized number as -0. This flush-to-zero treatment is applied before exception detection and special case handling.

Exceptional conditions are checked in the order given below. Execution of the instruction is terminated once any check detects an exceptional condition.

1 Disabled: an exception is raised if the FPU is disabled.

2 Invalid: an invalid operation is signaled if the input is a signaling NaN, or if this is a reciprocal square root of a number less than zero (including negative infinity and negative normalized/denormalized numbers, but excluding negative zero).

3 Divide-by-zero: a divide-by-zero is signaled if this is a reciprocal square root of zero (regardless of the sign of the zero).

4 Error: an FPU error is signaled if FPSCR.DN is 0 and the input is a positive denormalized number.

5 Inexact: this is an approximate instruction and inexact is signaled if this is a reciprocal square root of a positive normalized non-zero finite number. Inexact is not signaled if the input is a negative normalized number, a zero, an infinity, a denormalized number or a NaN. When inexact exceptions are requested by the user, an exception is always raised regardless of whether that condition arose. Overflow and underflow do not occur.

If the instruction does not raise an exception, a result is generated according to the following table. Where the behavior is not a special case, the instruction computes an approximate result using an implementation-dependent algorithm.

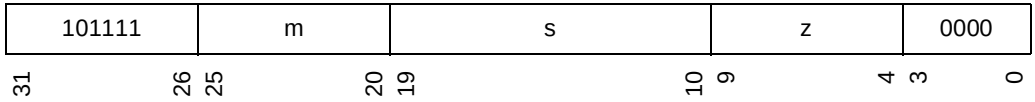
source1 →	+NORM	-NORM	+0	-0	+INF	-INF	+DNRM	-DNRM	qNaN	sNaN
	SRRA	qNaN	+INF	-INF	+0	qNaN	n/a	qNaN	qNaN	qNaN

FPU error is indicated by heavy shading and always raises an exception. Invalid operations and divide-by-zero are indicated by light shading and raise an exception if enabled. FPU disabled and inexact cases are not shown.

The normal ‘SRRA’ case uses an implementation-specific algorithm to calculate an approximation of the reciprocal square root of source1.

FST.D Rm, disp, DRz

FST.D Rm, disp, DRz




```
sr ← ZeroExtend64(SR);
base ← ZeroExtend64(Rm);
disp ← SignExtend10(s) << 3;
value ← FloatValue64(DRz);
IF (FpuIsDisabled(sr))
   THROW FPUDIS;
address ← ZeroExtend64(base + disp);
WriteMemory64(address, value);
```

Description:

This floating-point instruction stores a double-precision floating-point register to memory using register plus scaled immediate addressing. The effective address is formed by multiplying the sign-extended 10-bit immediate s by 8, and adding it to R_m. The 64-bit value of DR_z is written to this effective address.

This instruction is not considered an arithmetic operation, and it does not signal invalid operations. This instruction places no interpretation on the value transferred.

Possible exceptions:

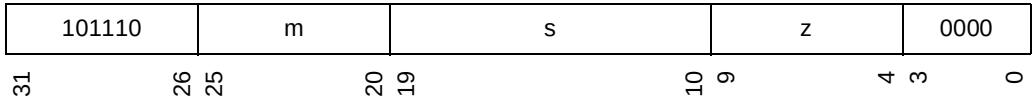
FPUDIS, WADDERR, WTLBMISS, WRITEPROT

Notes:

The ‘disp’ in the assembly syntax represents the immediate s after sign extension and scaling.

FST.P Rm, disp, FPz

FST.P Rm, disp, FPz




```
sr ← ZeroExtend64(SR);
base ← ZeroExtend64(Rm);
disp ← SignExtend10(s) << 3;
value ← FloatValuePair32(FPz);
IF (FpuIsDisabled(sr))
   THROW FPUDIS;
address ← ZeroExtend64(base + disp);
WriteMemoryPair32(address, value);
```

Description:

This floating-point instruction stores a pair of single-precision floating-point registers to memory using register plus scaled immediate addressing. The effective address (EA) is formed by multiplying the sign-extended 10-bit immediate s by 8, and adding it to R_m. The 64 bits of data in FP_z are written to the effective address as a pair of single-precision floating-point values. The 32 bits of data written to EA are from FR_f, and the 32 bits of data written to EA+4 are from FR_{f+1}.

This instruction is not considered an arithmetic operation, and it does not signal invalid operations. This instruction places no interpretation on the value transferred.

Possible exceptions:

FPUDIS, WADDERR, WTLBMISS, WRITEPROT

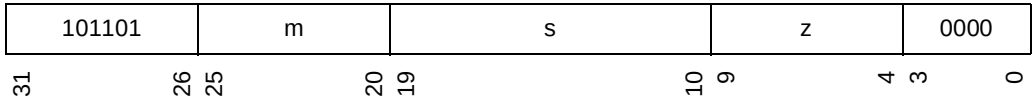
Notes:

The 'disp' in the assembly syntax represents the immediate s after sign extension and scaling.

The memory representation of pairs of single-precision floating-point registers is defined in *Volume 1, Chapter 3: Data representation*.

FST.S Rm, disp, FRz

FST.S Rm, disp, FRz




```
sr ← ZeroExtend64(SR);
base ← ZeroExtend64(Rm);
disp ← SignExtend10(s) << 2;
value ← FloatValue32(FRz);
IF (FpuIsDisabled(sr))
   THROW FPUDIS;
address ← ZeroExtend64(base + disp);
WriteMemory32(address, value);
```

Description:

This floating-point instruction stores a single-precision floating-point register to memory using register plus scaled immediate addressing. The effective address is formed by multiplying the sign-extended 10-bit immediate s by 4, and adding it to R_m. The 32-bit value of FR_z is written to the effective address.

This instruction is not considered an arithmetic operation, and it does not signal invalid operations. This instruction places no interpretation on the value transferred.

Possible exceptions:

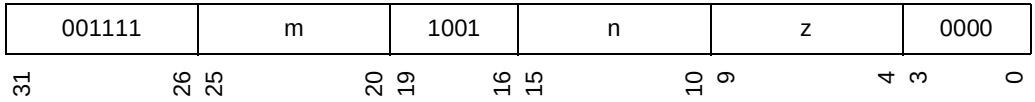
FPUDIS, WADDERR, WTLBMISS, WRITEPROT

Notes:

The ‘disp’ in the assembly syntax represents the immediate s after sign extension and scaling.

FSTX.D Rm, Rn, DRz

FSTX.D Rm, Rn, DRz




```
sr ← ZeroExtend64(SR);
base ← ZeroExtend64(Rm);
index ← SignExtend64(Rn);
value ← FloatValue64(DRz);
IF (FpuIsDisabled(sr))
   THROW FPUDIS;
address ← ZeroExtend64(base + index);
WriteMemory64(address, value);
```

Description:

This floating-point instruction stores a double-precision floating-point register to memory using register plus register addressing. The effective address is formed by adding R_m to R_n. The 64-bit value of DR_z is written to this effective address.

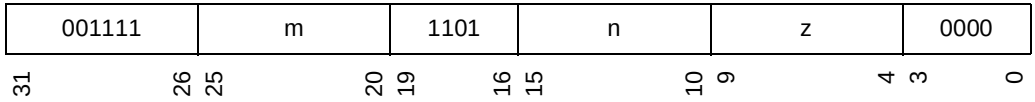
This instruction is not considered an arithmetic operation, and it does not signal invalid operations. This instruction places no interpretation on the value transferred.

Possible exceptions:

FPUDIS, WADDERR, WTLBMISS, WRITEPROT

FSTX.P Rm, Rn, FPz

FSTX.P Rm, Rn, FPz




```
sr ← ZeroExtend64(SR);
base ← ZeroExtend64(Rm);
index ← SignExtend64(Rn);
value ← FloatValuePair32(FPz);
IF (FpuIsDisabled(sr))
   THROW FPUDIS;
address ← ZeroExtend64(base + index);
WriteMemoryPair32(address, value);
```

Description:

This floating-point instruction stores a pair of single-precision floating-point registers to memory using register plus register addressing. The effective address (EA) is formed by adding R_m to R_n. The 64 bits of data in FP_z are written to the effective address as a pair of single-precision floating-point values. The 32 bits of data written to EA are from FR_f, and the 32 bits of data written to EA+4 are from FR_{f+1}.

This instruction is not considered an arithmetic operation, and it does not signal invalid operations. This instruction places no interpretation on the value transferred.

Possible exceptions:

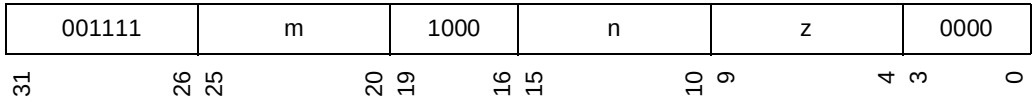
FPUDIS, WADDERR, WTLBMISS, WRITEPROT

Notes:

The memory representation of pairs of single-precision floating-point registers is defined in *Volume 1, Chapter 3: Data representation*.

FSTX.S Rm, Rn, FRz

FSTX.S Rm, Rn, FRz




```
sr ← ZeroExtend64(SR);
base ← ZeroExtend64(Rm);
index ← SignExtend64(Rn);
value ← FloatValue32(FRz);
IF (FpuIsDisabled(sr))
   THROW FPUDIS;
address ← ZeroExtend64(base + index);
WriteMemory32(address, value);
```

Description:

This floating-point instruction stores a single-precision floating-point register to memory using register plus register addressing. The effective address is formed by adding R_m to R_n. The 32-bit value of FR_z is written to the effective address.

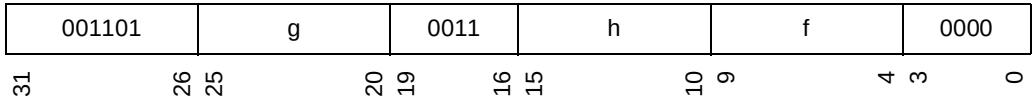
This instruction is not considered an arithmetic operation, and it does not signal invalid operations. This instruction places no interpretation on the value transferred.

Possible exceptions:

FPUDIS, WADDERR, WTLBMISS, WRITEPROT

FSUB.D DRg, DRh, DRf

FSUB.D DRg, DRh, DRf




```
sr ← ZeroExtend64(SR);
fps ← ZeroExtend32(FPSCR);
source1 ← FloatValue64(DRg);
source2 ← FloatValue64(DRh);
IF (FpuIsDisabled(sr))
   THROW FPUDIS;
result, fps ← FSUB_D(source1, source2, fps);
IF (FpuEnableV(fps) AND FpuCauseV(fps))
   THROW FPUEXC, fps;
IF (FpuCauseE(fps))
   THROW FPUEXC, fps;
IF ((FpuEnableI(fps) OR FpuEnableO(fps)) OR FpuEnableU(fps))
   THROW FPUEXC, fps;
DRf ← FloatRegister64(result);
FPSCR ← Register(fps);
```

Description:

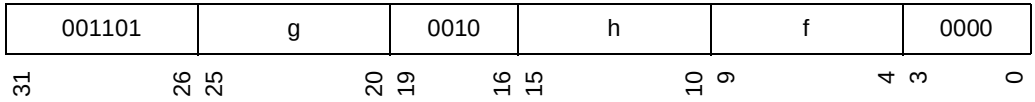
This floating-point instruction performs a double-precision floating-point subtraction. It subtracts DR_h from DR_g and places the result in DR_f. The rounding mode is determined by FPSCR.RM.

Possible exceptions:

FPUDIS, FPUEXC

FSUB.S FRg, FRh, FRf

FSUB.S FRg, FRh, FRf




```
sr ← ZeroExtend64(SR);
fps ← ZeroExtend32(FPSCR);
source1 ← FloatValue32(FRg);
source2 ← FloatValue32(FRh);
IF (FpuIsDisabled(sr))
   THROW FPUDIS;
result, fps ← FSUB_S(source1, source2, fps);
IF (FpuEnableV(fps) AND FpuCauseV(fps))
   THROW FPUEXC, fps;
IF (FpuCauseE(fps))
   THROW FPUEXC, fps;
IF ((FpuEnableI(fps) OR FpuEnableO(fps)) OR FpuEnableU(fps))
   THROW FPUEXC, fps;
FRf ← FloatRegister32(result);
FPSCR ← Register(fps);
```

Description:

This floating-point instruction performs a single-precision floating-point subtraction. It subtracts FR_h from FR_g and places the result in FR_f. The rounding mode is determined by FPSCR.RM.

Possible exceptions:

FPUDIS, FPUEXC

FSUB.S and FSUB.D special cases:

When FPSCR.DN is 1, a positive denormalized number is treated as +0 and a negative denormalized number as -0. This flush-to-zero treatment is applied before exception detection and special case handling.

Exceptional conditions are checked in the order given below. Execution of the instruction is terminated once any check detects an exceptional condition.

1 Disabled: an exception is raised if the FPU is disabled.

2 Invalid: an invalid operation is signaled if either input is a signaling NaN, or if the inputs are similarly signed infinities.

3 Error: an FPU error is signaled if FPSCR.DN is 0, neither input is a NaN and either input is a denormalized number.

4 Inexact, underflow and overflow: these are checked together and can be signaled in combination. When inexact, underflow or overflow exceptions are requested by the user, an exception is always raised regardless of whether that condition arose.

If the instruction does not raise an exception, a result is generated according to the following table.

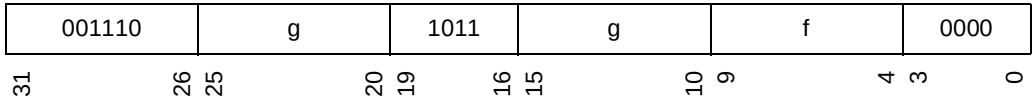
source1 → ↓ source2	+NORM, -NORM	+0	-0	+INF	-INF	+DNRM, -DNRM	qNaN	sNaN
+,-NORM	SUB	SUB	SUB	+INF	-INF	n/a	qNaN	qNaN
+0	source1	+0	-0	+INF	-INF	n/a	qNaN	qNaN
-0	source1	+0	+0	+INF	-INF	n/a	qNaN	qNaN
+INF	-INF	-INF	-INF	qNaN	-INF	n/a	qNaN	qNaN
-INF	+INF	+INF	+INF	+INF	qNaN	n/a	qNaN	qNaN
+, -DNRM	n/a	n/a	n/a	n/a	n/a	n/a	qNaN	qNaN
qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN
sNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN

FPU error is indicated by heavy shading and always raises an exception. Invalid operations are indicated by light shading and raise an exception if enabled. FPU disabled, inexact, underflow and overflow cases are not shown.

The behavior of the normal 'SUB' case is described by the IEEE754 specification.

FTRC.DL DRg, FRf

FTRC.DL DRg, FRf




```
sr ← ZeroExtend32(SR);
fps ← ZeroExtend32(FPSCR);
source ← FloatValue64(DRg);
IF (FpuIsDisabled(sr))
   THROW FPUDIS;
result, fps ← FTRC_DL(source, fps);
IF (FpuEnableV(fps) AND FpuCauseV(fps))
   THROW FPUEXC, fps;
FRf ← FloatRegister32(result);
FPSCR ← Register(fps);
```

Description:

This floating-point instruction performs a double-precision floating-point to signed 32-bit integer conversion. It reads a double-precision value from DR_g, converts it to a signed 32-bit integral range and places the result in FR_f. The conversion is achieved by rounding to zero (truncation) with saturation to the limits of the target signed integral range. The value of FPSCR.RM is ignored.

In order to perform integer operations on the result, it needs to be subsequently moved to the general-purpose register file.

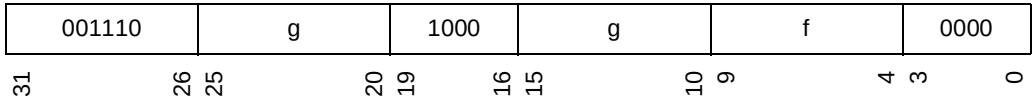
Both source operand fields must be encoded with the source register designator g.

Possible exceptions:

FPUDIS, FPUEXC

FTRC.SL FRg, FRf

FTRC.SL FRg, FRf




```
sr ← ZeroExtend32(SR);
fps ← ZeroExtend32(FPSCR);
source ← FloatValue32(FRg);
IF (FpuIsDisabled(sr))
   THROW FPUDIS;
result, fps ← FTRC_SL(source, fps);
IF (FpuEnableV(fps) AND FpuCauseV(fps))
   THROW FPUEXC, fps;
FRf ← FloatRegister32(result);
FPSCR ← Register(fps);
```

Description:

This floating-point instruction performs a single-precision floating-point to signed 32-bit integer conversion. It reads a single-precision value from FR_g, converts it to a signed 32-bit integral range and places the result in FR_f. The conversion is achieved by rounding to zero (truncation) with saturation to the limits of the target signed integral range. The value of FPSCR.RM is ignored.

In order to perform integer operations on the result, it needs to be subsequently moved to the general-purpose register file.

Both source operand fields must be encoded with the source register designator g.

Possible exceptions:

FPUDIS, FPUEXC

FTRC.SL and FTRC.DL special cases:

Regardless of FPSCR.DN, denormalized numbers are treated as 0. These instructions do not cause FPU Error.

Exceptional conditions are checked in the order given below. Execution of the instruction is terminated once any check detects an exceptional condition.

1 Disabled: an exception is raised if the FPU is disabled.

2 Invalid: an invalid operation is signaled if the conversion overflows the target range. This is caused by out-of-range normalized numbers, infinities and NaNs.

If the instruction does not raise an exception, a result is generated according to the following table.

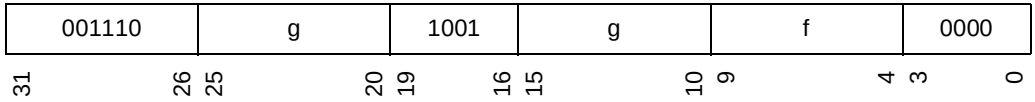
source1 →	+NORM, -NORM (in range)	+0	-0	+INF or +NORM (out of range)	-INF or -NORM (out of range)	+DNRM, -DNRM	qNaN	sNaN
TRC	0		0	$+2^{31} - 1$	-2^{31}	0	-2^{31}	-2^{31}

Invalid operations are indicated by light shading and raise an exception if enabled. FPU disabled cases are not shown.

The behavior of the normal 'TRC' case is described by the IEEE754 specification, though only the round to zero rounding mode is supported by this instruction.

FTRC.DQ DRg, DRf

FTRC.DQ DRg, DRf




```
sr ← ZeroExtend32(SR);
fps ← ZeroExtend32(FPSCR);
source ← FloatValue64(DRg);
IF (FpuIsDisabled(sr))
   THROW FPUDIS;
result, fps ← FTRC_DQ(source, fps);
IF (FpuEnableV(fps) AND FpuCauseV(fps))
   THROW FPUEXC, fps;
DRf ← FloatRegister64(result);
FPSCR ← Register(fps);
```

Description:

This floating-point instruction performs a double-precision floating-point to signed 64-bit integer conversion. It reads a double-precision value from DR_g, converts it to a signed 64-bit integral range and places the result in DR_f. The conversion is achieved by rounding to zero (truncation) with saturation to the limits of the target signed integral range. The value of FPSCR.RM is ignored.

In order to perform integer operations on the result, it needs to be subsequently moved to the general-purpose register file.

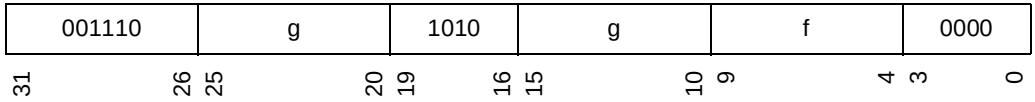
Both source operand fields must be encoded with the source register designator g.

Possible exceptions:

FPUDIS, FPUEXC

FTRC.SQ FRg, DRf

FTRC.SQ FRg, DRf




```
sr ← ZeroExtend32(SR);
fps ← ZeroExtend32(FPSCR);
source ← FloatValue32(FRg);
IF (FpuIsDisabled(sr))
   THROW FPUDIS;
result, fps ← FTRC_SQ(source, fps);
IF (FpuEnableV(fps) AND FpuCauseV(fps))
   THROW FPUEXC, fps;
DRf ← FloatRegister64(result);
FPSCR ← Register(fps);
```

Description:

This floating-point instruction performs a single-precision floating-point to signed 64-bit integer conversion. It reads a single-precision value from FR_g, converts it to a signed 64-bit integral range and places the result in DR_f. The conversion is achieved by rounding to zero (truncation) with saturation to the limits of the target signed integral range. The value of FPSCR.RM is ignored.

In order to perform integer operations on the result, it needs to be subsequently moved to the general-purpose register file.

Both source operand fields must be encoded with the source register designator g.

Possible exceptions:

FPUDIS, FPUEXC

FTRC.SQ and FTRC.DQ special cases:

Regardless of FPSCR.DN, denormalized numbers are treated as 0. These instructions do not cause FPU Error.

Exceptional conditions are checked in the order given below. Execution of the instruction is terminated once any check detects an exceptional condition.

1 Disabled: an exception is raised if the FPU is disabled.

2 Invalid: an invalid operation is signaled if the conversion overflows the target range. This is caused by out-of-range normalized numbers, infinities and NaNs.

If the instruction does not raise an exception, a result is generated according to the following table.

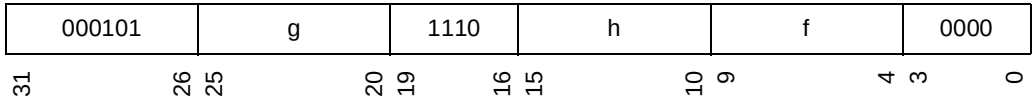
source1 →	+NORM, -NORM (in range)	+0	-0	+INF or +NORM (out of range)	-INF or -NORM (out of range)	+DNRM, -DNRM	qNaN	sNaN
TRC	0	0		$+2^{63} - 1$	-2^{63}	0	-2^{63}	-2^{63}

Invalid operations are indicated by light shading and raise an exception if enabled. FPU disabled cases are not shown.

The behavior of the normal 'TRC' case is described by the IEEE754 specification, though only the round to zero rounding mode is supported by this instruction.

FTRV.S MTRXg, FVh, FVf

FTRV.S MTRXg, FVh, FVf




```
sr ← ZeroExtend64(SR);
fps ← ZeroExtend32(FPSCR);
source1 ← FloatValueMatrix32(MTRXg);
source2 ← FloatValueVector32(FVh);
IF (FpuIsDisabled(sr))
   THROW FPUDIS;
result, fps ← FTRV_S(source1, source2, fps);
IF (((FpuEnableV(fps) OR FpuEnableI(fps)) OR FpuEnableO(fps)) OR FpuEnableU(fps))
   THROW FPUEXC, fps;
FVf ← FloatRegisterVector32(result);
FPSCR ← Register(fps);
```

Description:

This floating-point instruction multiplies a matrix, $MTRX_g$, with a vector, FV_h, and places the resulting vector in FV_f. The matrix contains sixteen single-precision floating-point values. The vector contains four single-precision floating-point values. The matrix-vector multiplication is specified as:

$$FR_{f+0} = \sum_{i=0}^{3} FR_{g+i\times 4} \times FR_{h+i}$$

$$FR_{f+1} = \sum_{i=0}^{3} FR_{g+1+i\times 4} \times FR_{h+i}$$

$$FR_{f+2} = \sum_{i=0}^{3} FR_{g+2+i\times 4} \times FR_{h+i}$$

$$FR_{f+3} = \sum_{i=0}^{3} FR_{g+3+i\times 4} \times FR_{h+i}$$

This is an approximate computation. The specified error in the result value is defined in *Volume 1, Chapter 8: SHmedia floating-point*.

Possible exceptions:

FPUDIS, FPUEXC

FTRV.S special cases:

FTRV.S is an approximate instruction. Denormalized numbers are supported:

- When FPSCR.DN is 0, denormalized numbers are treated as their denormalized value in the FTRV.S calculation. This instruction never signals an FPU error.
- When FPSCR.DN is 1, a positive denormalized number is treated as +0 and a negative denormalized number as -0. This flush-to-zero treatment is applied before exception detection and special case handling.

Exceptional conditions are checked in the order given below. Execution of the instruction is terminated once any check detects an exceptional condition.

1 Disabled: an exception is raised if the FPU is disabled.

2 Invalid: an invalid operation is signaled if any of the inputs is a signaling NaN, there is a multiplication of a zero by an infinity, or there is an addition of differently signed infinities where none of the inputs is a qNaN.

  The multiplication is performed with sufficient precision to avoid overflow, and therefore the multiplication of any two finite numbers does not produce an infinity. The multiplication result will be an infinity only if there is a multiplication of an infinity with a normalized number, an infinity with a denormalized number or an infinity with an infinity.

  The addition of differently signed infinities is detected if there is (at least) one positive infinity and (at least) one negative infinity in the set of 4 multiplication results in any of the 4 inner-products calculated by this instruction.

  This instruction does not check all of its inputs for invalid operations and then raise an exception accordingly. If invalid operation exceptions are requested by the user, this instruction always raises that exception. If this exception is not requested by the user, then each of the four inner-products is checked separately for an invalid operation (as described above) and the appropriate result is set to qNaN for each inner-product that is invalid.

3 Inexact, underflow and overflow: these are checked together and can be signaled in combination. This is an approximate instruction and inexact is signaled except where special cases occur. Precise details of the approximate transform algorithm, including the detection of underflow and overflow cases, are implementation dependent. When inexact, underflow or overflow exceptions are requested by the user, an exception is always raised regardless of whether that condition arose.

If the instruction does not raise an exception, results are generated according to the following tables. The special case tables are applied separately with the appropriate vector operands to each of the four inner-products calculated by this instruction. Each of the 4 pairs of multiplication operands (source1 and source2) is selected from corresponding elements of the two 4-element source vectors and multiplied:

source1 → ↓ source2	+,-NORM, +,-DNRM	+0	-0	+INF	-INF	qNaN	sNaN
+,-NORM and +,-DNRM	FTRVMUL	+0, -0	-0, +0	+INF, -INF	-INF, +INF	qNaN	qNaN
+0	+0, -0	+0	-0	qNaN	qNaN	qNaN	qNaN
-0	-0, +0	-0	+0	qNaN	qNaN	qNaN	qNaN
+INF	+INF, -INF	qNaN	qNaN	+INF	-INF	qNaN	qNaN
-INF	-INF, +INF	qNaN	qNaN	-INF	+INF	qNaN	qNaN
qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN
sNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN

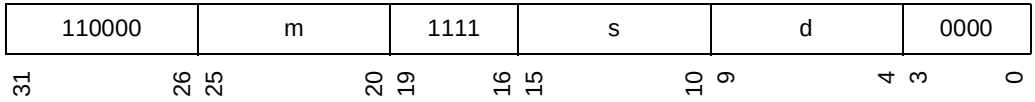
If any of the multiplications evaluates to qNaN, then the result of the instruction is qNaN and no further analysis need be performed. In the 'FTRVMUL', +0, -0, +INF and -INF cases, the 4 addition operands (labeled temp0 to temp3) are summed:

	temp0 →	FTRVMUL, +0, -0			+INF			-INF		
↓ temp2	temp1→ ↓ temp3	FTRVMUL, +0, -0	+INF	-INF	FTRVMUL, +0, -0	+INF	-INF	FTRVMUL, +0, -0	+INF	-INF
FTRVMUL, +0, -0	FTRVMUL, +0, -0	FTRVADD	+INF	-INF	+INF	+INF	qNaN	-INF	qNaN	-INF
	+INF	+INF	+INF	qNaN	+INF	+INF	qNaN	qNaN	qNaN	qNaN
	-INF	-INF	qNaN	-INF	qNaN	qNaN	qNaN	-INF	qNaN	-INF
+INF	FTRVMUL, +0, -0	+INF	+INF	qNaN	+INF	+INF	qNaN	qNaN	qNaN	qNaN
	+INF	+INF	+INF	qNaN	+INF	+INF	qNaN	qNaN	qNaN	qNaN
	-INF	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN
-INF	FTRVMUL, +0, -0	-INF	qNaN	-INF	qNaN	qNaN	qNaN	-INF	qNaN	-INF
	+INF	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN	qNaN
	-INF	-INF	qNaN	-INF	qNaN	qNaN	qNaN	-INF	qNaN	-INF

Inexact is signaled in the 'FTRVADD' case. Exception cases are not indicated by shading for this instruction. Where the behavior is not a special case, the instruction computes an approximate result using an implementation-dependent algorithm.

GETCFG Rm, disp, Rd

GETCFG Rm, disp, Rd




```
md ← ZeroExtend1(MD);
base ← ZeroExtend64(Rm);
disp ← SignExtend6(s);
index ← ZeroExtend64(base + disp);
IF (md = 0)
   THROW RESINST;
IF (IsUndefinedConfigurationRegister(index))
   result ← UNDEFINED;
ELSE
   result ← ReadConfigurationRegister(index);
Rd ← Register(result);
```

Description:

This instruction copies a configuration register to R_d. The source configuration register is identified by adding R_m to the sign-extended 6-bit immediate s.

GETCFG is a privileged instruction.

A read from an undefined configuration register gives an architecturally-undefined result. Note that configuration registers do not, in general, have simple read/write semantics.

Possible exceptions:

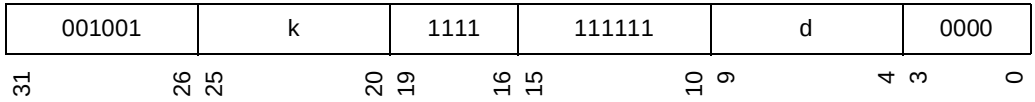
RESINST

Notes:

The ‘disp’ in the assembly syntax represents the immediate s after sign extension.

GETCON CRk, Rd

GETCON CRk, Rd




```
md ← ZeroExtend1(MD);
index ← ZeroExtend6(k);
IF ((md = 0) AND IsPrivilegedControlRegister(index))
   THROW RESINST;
IF (IsUndefinedControlRegister(index))
   result ← UNDEFINED;
ELSE
   result ← ReadControlRegister(index);
Rd ← Register(result);
```

Description:

This instruction copies CR_k to R_d.

GETCON from a privileged control register is a privileged instruction. GETCON from a user-accessible control register is not a privileged instruction.

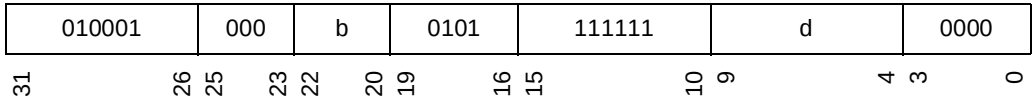
A read from an undefined control register gives an architecturally-undefined result. Note that control registers do not, in general, have simple read/write semantics.

Possible exceptions:

RESINST

GETTR TRb, Rd

GETTR TRb, Rd



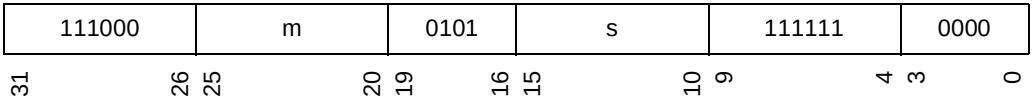

```
target ← ZeroExtend64(TRb);
result ← target;
Rd ← Register(result);
```

Description:

This instruction copies the value held in the target register TR_b to R_d. The value returned by GETTR ensures that any unimplemented higher bits of the source target register are seen as sign extensions of the highest implemented bit.

ICBI Rm, disp

ICBI Rm, disp




```
base ← ZeroExtend64(Rm);
disp ← SignExtend6(s) << 5;
address ← ZeroExtend64(base + disp);
IF (MalformedAddress(address))
   THROW IADDERR, address;
IF (MMU() AND InstInvalidateMiss(address))
   THROW ITLBMISS, address;
IF (NOT (MMU() AND ExecuteProhibited(address)))
   ICBI(address);
```

Description:

This instruction invalidates an instruction cache block (if any) that corresponds to a specified effective address. If a unified cache organization is used and the data in the instruction cache block is dirty, it is discarded without write-back to memory.

The effective address is calculated by adding R_m to the sign-extended 6-bit immediate s multiplied by 32. The scaling factor is fixed at 32 regardless of the cache block size. There is no misalignment check on this instruction, and the calculated effective address can be any byte address. The calculated effective address is automatically aligned downwards to the nearest exact multiple of the cache block size. The effective address identifies a surrounding block of memory, which starts at an address aligned to the cache block size and has a size equal to the cache block size. The cache block size is implementation dependent.

ICBI checks for address error and raises an IADDERR exception if this check fails. The implementation then determines whether there is an entry in the instruction cache for this ICBI to invalidate. Some implementations can perform a translation look-up to determine this; those implementations will raise an ITLBMISS exception if there is no translation available. Other implementations are able to determine this without ever raising ITLBMISS. Thus, whether an ITLBMISS exception can be raised by ICBI is implementation dependent.

If there is no entry in the instruction cache for this ICBI to invalidate, then the ICBI can complete since no invalidation is required. If there is an entry in the instruction cache for this ICBI to invalidate, then a check is made for protection violation. If a protection violation occurs, the instruction executes to completion without exception launch, but does not affect the state of the instruction cache.

Explicit synchronization instructions are required to synchronize the effects of cache coherency instructions. SYNCI must be used to guarantee that previous ICBI instructions have completed their invalidation on the instruction cache.

After completion, assuming no exception was raised and assuming that no protection violation was discarded, it is guaranteed that the targeted memory block in effective address space is not present in any instruction or unified cache.

Note that ICBI performs invalidation on effective addresses. There is no guarantee of invalidation of aliases at other effective addresses or in other effective address spaces.

The behavior of this instruction when the MMU is disabled is described in *Volume 1, Chapter 6: SHmedia memory instructions*.

Possible exceptions:

IADDERR, ITLBMISS

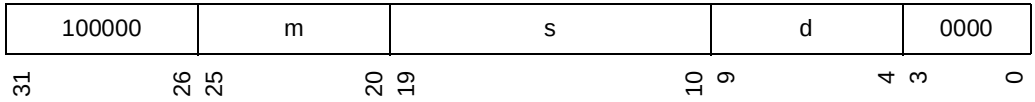
Notes:

For correct operation, software must exercise care when using ICBI.

The ‘disp’ in the assembly syntax represents the immediate s after sign extension and scaling.

LD.B Rm, disp, Rd

LD.B Rm, disp, Rd




```
d_field ← ZeroExtend6(d);
base ← ZeroExtend64(Rm);
disp ← SignExtend10(s);
address ← ZeroExtend64(base + disp);
IF (d_field = 63)
   result ← PrefetchMemory(address);
ELSE
   result ← SignExtend8(ReadMemory8(address));
Rd ← Register(result);
```

Description:

This instruction loads a byte from the effective address formed by adding R_m to the sign-extended 10-bit immediate s. If the destination register is not $R_{63,}$ the loaded byte is sign-extended and placed in R_d. In exceptional cases, an appropriate exception is raised.

If the destination register is R_{63}, this indicates a software-directed data prefetch from the specified effective address. Software can use this instruction to give advance notice that particular data will be required. It is implementation-specific as to whether a prefetch will be performed. The effective address identifies a surrounding block of memory, which starts at an address aligned to the cache block size and has a size equal to the cache block size. The cache block size is implementation dependent. In exceptional cases, no exception is raised and the prefetch has no effect.

Possible exceptions:

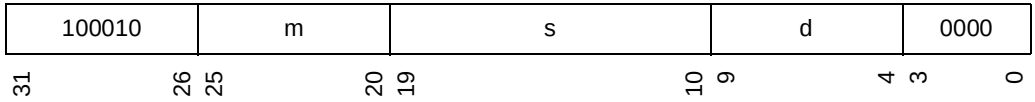
RADDERR, RTLBMISS, READPROT

Notes:

The ‘disp’ in the assembly syntax represents the immediate s after sign extension.

LD.L Rm, disp, Rd

LD.L Rm, disp, Rd




```
d_field ← ZeroExtend6(d);
base ← ZeroExtend64(Rm);
disp ← SignExtend10(s) << 2;
address ← ZeroExtend64(base + disp);
IF (d_field = 63)
   result ← PrefetchMemory(address);
ELSE
   result ← SignExtend32(ReadMemory32(address));
Rd ← Register(result);
```

Description:

This instruction loads a long-word from the effective address formed by adding R_m to the sign-extended 10-bit immediate s multiplied by 4. If the destination register is not $R_{63,}$ the loaded long-word is sign-extended and placed in R_d. Note that only one version of this instruction is provided: the representation of signed and unsigned 32-bit data in a register is the same. In exceptional cases, including misaligned loads, an appropriate exception is raised.

If the destination register is R_{63}, this indicates a software-directed data prefetch from the specified effective address. Software can use this instruction to give advance notice that particular data will be required. It is implementation-specific as to whether a prefetch will be performed. The effective address identifies a surrounding block of memory, which starts at an address aligned to the cache block size and has a size equal to the cache block size. The cache block size is implementation dependent. In exceptional cases, no exception is raised and the prefetch has no effect.

Possible exceptions:

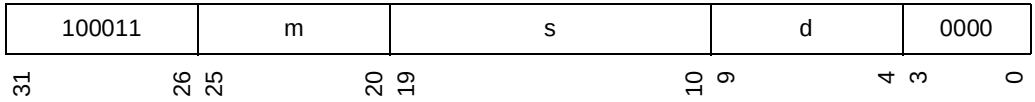
RADDERR, RTLBMISS, READPROT

Notes:

The ‘disp’ in the assembly syntax represents the immediate s after sign extension and scaling.

LD.Q Rm, disp, Rd

LD.Q Rm, disp, Rd




```
d_field ← ZeroExtend6(d);
base ← ZeroExtend64(Rm);
disp ← SignExtend10(s) << 3;
address ← ZeroExtend64(base + disp);
IF (d_field = 63)
   result ← PrefetchMemory(address);
ELSE
   result ← ZeroExtend64(ReadMemory64(address));
Rd ← Register(result);
```

Description:

This instruction loads a quad-word from the effective address formed by adding R_m to the sign-extended 10-bit immediate s multiplied by 8. If the destination register is not $R_{63,}$ the loaded quad-word is placed in R_d. Note that only one version of this instruction is provided: sign is unimportant when loading an object of the same size as a register. In exceptional cases, including misaligned loads, an appropriate exception is raised.

If the destination register is R_{63}, this indicates a software-directed data prefetch from the specified effective address. Software can use this instruction to give advance notice that particular data will be required. It is implementation-specific as to whether a prefetch will be performed. The effective address identifies a surrounding block of memory, which starts at an address aligned to the cache block size and has a size equal to the cache block size. The cache block size is implementation dependent. In exceptional cases, no exception is raised and the prefetch has no effect.

Possible exceptions:

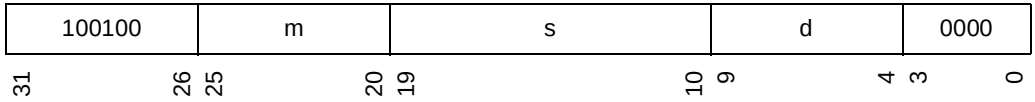
RADDERR, RTLBMISS, READPROT

Notes:

The ‘disp’ in the assembly syntax represents the immediate s after sign extension and scaling.

LD.UB Rm, disp, Rd

LD.UB Rm, disp, Rd




```
d_field ← ZeroExtend6(d);
base ← ZeroExtend64(Rm);
disp ← SignExtend10(s);
address ← ZeroExtend64(base + disp);
IF (d_field = 63)
   result ← PrefetchMemory(address);
ELSE
   result ← ZeroExtend8(ReadMemory8(address));
Rd ← Register(result);
```

Description:

This instruction loads a byte from the effective address formed by adding R_m to the sign-extended 10-bit immediate s. If the destination register is not $R_{63,}$ the loaded byte is zero-extended and placed in R_d. In exceptional cases, an appropriate exception is raised.

If the destination register is R_{63}, this indicates a software-directed data prefetch from the specified effective address. Software can use this instruction to give advance notice that particular data will be required. It is implementation-specific as to whether a prefetch will be performed. The effective address identifies a surrounding block of memory, which starts at an address aligned to the cache block size and has a size equal to the cache block size. The cache block size is implementation dependent. In exceptional cases, no exception is raised and the prefetch has no effect.

Possible exceptions:

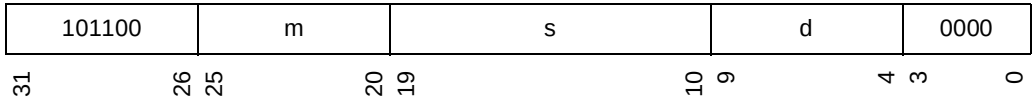
RADDERR, RTLBMISS, READPROT

Notes:

The ‘disp’ in the assembly syntax represents the immediate s after sign extension.

LD.UW Rm, disp, Rd

LD.UW Rm, disp, Rd




```
d_field ← ZeroExtend6(d);
base ← ZeroExtend64(Rm);
disp ← SignExtend10(s) << 1;
address ← ZeroExtend64(base + disp);
IF (d_field = 63)
   result ← PrefetchMemory(address);
ELSE
   result ← ZeroExtend16(ReadMemory16(address));
Rd ← Register(result);
```

Description:

This instruction loads a word from the effective address formed by adding R_m to the sign-extended 10-bit immediate s multiplied by 2. If the destination register is not $R_{63,}$ the loaded word is zero-extended and placed in R_d. In exceptional cases, including misaligned loads, an appropriate exception is raised.

If the destination register is R_{63}, this indicates a software-directed data prefetch from the specified effective address. Software can use this instruction to give advance notice that particular data will be required. It is implementation-specific as to whether a prefetch will be performed. The effective address identifies a surrounding block of memory, which starts at an address aligned to the cache block size and has a size equal to the cache block size. The cache block size is implementation dependent. In exceptional cases, no exception is raised and the prefetch has no effect.

Possible exceptions:

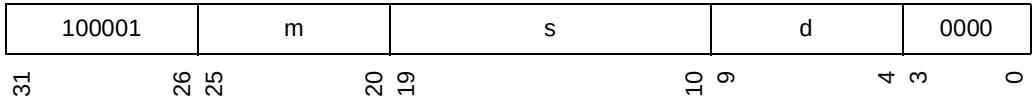
RADDERR, RTLBMISS, READPROT

Notes:

The ‘disp’ in the assembly syntax represents the immediate s after sign extension and scaling.

LD.W Rm, disp, Rd

LD.W Rm, disp, Rd




```
d_field ← ZeroExtend6(d);
base ← ZeroExtend64(Rm);
disp ← SignExtend10(s) << 1;
address ← ZeroExtend64(base + disp);
IF (d_field = 63)
   result ← PrefetchMemory(address);
ELSE
   result ← SignExtend16(ReadMemory16(address));
Rd ← Register(result);
```

Description:

This instruction loads a word from the effective address formed by adding R_m to the sign-extended 10-bit immediate s multiplied by 2. If the destination register is not $R_{63,}$ the loaded word is sign-extended and placed in $R_{d.}$ In exceptional cases, including misaligned loads, an appropriate exception is raised.

If the destination register is R_{63}, this indicates a software-directed data prefetch from the specified effective address. Software can use this instruction to give advance notice that particular data will be required. It is implementation-specific as to whether a prefetch will be performed. The effective address identifies a surrounding block of memory, which starts at an address aligned to the cache block size and has a size equal to the cache block size. The cache block size is implementation dependent. In exceptional cases, no exception is raised and the prefetch has no effect.

Possible exceptions:

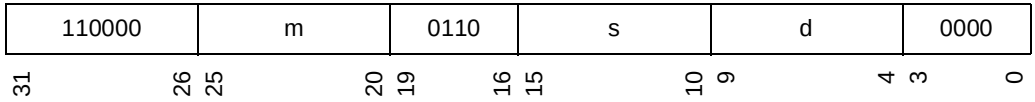
RADDERR, RTLBMISS, READPROT

Notes:

The ‘disp’ in the assembly syntax represents the immediate s after sign extension and scaling.

LDHI.L Rm, disp, Rd

LDHI.L Rm, disp, Rd




```
base ← ZeroExtend64(Rm);
disp ← SignExtend6(s);
address ← base + disp;
bytecount ← (address ∧ 0x3) + 1;
bitcount ← bytecount × 8;
shift ← ZeroExtend5((~ (address ∧ 0x3)) × 8);
mem ← ZeroExtendbitcount(ReadMemoryHighbitcount(address));
IF (IsLittleEndian())
   result ← SignExtend32(mem << shift);
ELSE
   result ← ZeroExtend32(mem);
Rd ← Register(result);
```

Description:

This instruction loads the high part of a misaligned long-word from memory into R_d. The effective address is formed by adding the sign-extended 6-bit immediate s to R_m. The effective address points to the highest byte in the misaligned long-word. The address of the lowest byte in the high part of the misaligned long-word is determined by masking the least significant 2 bits of the effective address to 0.

This instruction loads the inclusive range of memory bytes starting at that lowest byte and ending at that highest byte. If the effective address is actually 4-byte aligned, then all 4 bytes are loaded. The loaded bytes are placed into the appropriate bytes within R_d, and other bytes are set to 0 or a sign-extension of bit 31 as required.

This instruction can be used in conjunction with LDLO.L and OR to load and sign-extend a misaligned long-word into a register. In this case, the LDHI.L effective address should be 3 bytes larger than the LDLO.L effective address.

Possible exceptions:

RADDERR, RTLBMISS, READPROT

LDHI.L Byte Mappings:

The mapping between byte locations in memory and byte positions in the destination register is shown below for each value of the low 2 bits in the effective address (EA). Each byte in the register is 0, a sign-extension or maps to the given memory address.

Little endian mode

Low 2 bits of EA↓	Bit 63 — Target register — Bit 0							
	Byte 7	**Byte 6**	**Byte 5**	**Byte 4**	**Byte 3**	**Byte 2**	**Byte 1**	**Byte 0**
0x0	sign extension of bit 31				EA-0	0	0	0
0x1	sign extension of bit 31				EA-0	EA-1	0	0
0x2	sign extension of bit 31				EA-0	EA-1	EA-2	0
0x3	sign extension of bit 31				EA-0	EA-1	EA-2	EA-3

Big endian mode

Low 2 bits of EA↓	Bit 63 — Target register — Bit 0							
	Byte 0	**Byte 1**	**Byte 2**	**Byte 3**	**Byte 4**	**Byte 5**	**Byte 6**	**Byte 7**
0x0	0	0	0	0	0	0	0	EA-0
0x1	0	0	0	0	0	0	EA-1	EA-0
0x2	0	0	0	0	0	EA-2	EA-1	EA-0
0x3	0	0	0	0	EA-3	EA-2	EA-1	EA-0

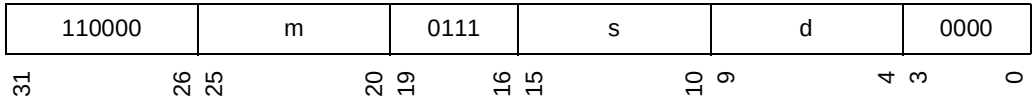
Notes:

The ‘disp’ in the assembly syntax represents the immediate s after sign extension.

When the memory access for LDHI.L causes an exception, the TEA control register is initialized with the effective address of the access. This corresponds to the address of the highest byte in the misaligned long-word.

LDHI.Q Rm, disp, Rd

LDHI.Q Rm, disp, Rd




```
base ← ZeroExtend64(Rm);
disp ← SignExtend6(s);
address ← base + disp;
bytecount ← (address ∧ 0x7) + 1;
bitcount ← bytecount × 8;
shift ← ZeroExtend6((~ (address ∧ 0x7)) × 8);
mem ← ZeroExtendbitcount(ReadMemoryHighbitcount(address));
IF (IsLittleEndian())
   result ← ZeroExtend64(mem << shift);
ELSE
   result ← ZeroExtend64(mem);
Rd ← Register(result);
```

Description:

This instruction loads the high part of a misaligned quad-word from memory into R_d. The effective address is formed by adding the sign-extended 6-bit immediate s to R_m. The effective address points to the highest byte in the misaligned quad-word. The address of the lowest byte in the high part of the misaligned quad-word is determined by masking the least significant 3 bits of the effective address to 0.

This instruction loads the inclusive range of memory bytes starting at that lowest byte and ending at that highest byte. If the effective address is actually 8-byte aligned, then all 8 bytes are loaded. The loaded bytes are placed into the appropriate bytes within R_d, and any other bytes are set to 0.

This instruction can be used in conjunction with LDLO.Q and OR to load a misaligned quad-word from memory into a register. In this case, the LDHI.Q effective address should be 7 bytes larger than the LDLO.Q effective address.

Possible exceptions:

RADDERR, RTLBMISS, READPROT

LDHI.Q Byte Mappings:

The mapping between byte locations in memory and byte positions in the destination register is shown below for each value of the low 3 bits in the effective address (EA). Each byte in the register is 0 or maps to the given memory address.

Little endian mode	Bit 63 Target register Bit 0							
Low 3 bits of EA↓	Byte 7	Byte 6	Byte 5	Byte 4	Byte 3	Byte 2	Byte 1	Byte 0
0x0	EA-0	0	0	0	0	0	0	0
0x1	EA-0	EA-1	0	0	0	0	0	0
0x2	EA-0	EA-1	EA-2	0	0	0	0	0
0x3	EA-0	EA-1	EA-2	EA-3	0	0	0	0
0x4	EA-0	EA-1	EA-2	EA-3	EA-4	0	0	0
0x5	EA-0	EA-1	EA-2	EA-3	EA-4	EA-5	0	0
0x6	EA-0	EA-1	EA-2	EA-3	EA-4	EA-5	EA-6	0
0x7	EA-0	EA-1	EA-2	EA-3	EA-4	EA-5	EA-6	EA-7

Big endian mode	Bit 63 Target register Bit 0							
Low 3 bits of EA↓	Byte 0	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7
0x0	0	0	0	0	0	0	0	EA-0
0x1	0	0	0	0	0	0	EA-1	EA-0
0x2	0	0	0	0	0	EA-2	EA-1	EA-0
0x3	0	0	0	0	EA-3	EA-2	EA-1	EA-0
0x4	0	0	0	EA-4	EA-3	EA-2	EA-1	EA-0

Big endian mode	Bit 63 Target register							Bit 0
Low 3 bits of EA↓	Byte 0	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7
0x5	0	0	EA-5	EA-4	EA-3	EA-2	EA-1	EA-0
0x6	0	EA-6	EA-5	EA-4	EA-3	EA-2	EA-1	EA-0
0x7	EA-7	EA-6	EA-5	EA-4	EA-3	EA-2	EA-1	EA-0

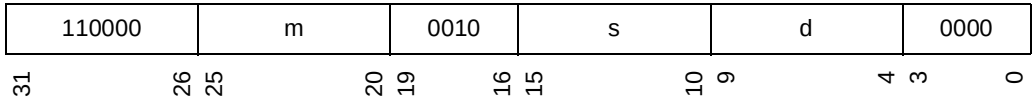
Notes:

The ‘disp’ in the assembly syntax represents the immediate s after sign extension.

When the memory access for LDHI.Q causes an exception, the TEA control register is initialized with the effective address of the access. This corresponds to the address of the highest byte in the misaligned quad-word.

LDLO.L Rm, disp, Rd

LDLO.L Rm, disp, Rd




```
base ← ZeroExtend64(Rm);
disp ← SignExtend6(s);
address ← ZeroExtend64(base + disp);
bytecount ← 4 - (address ∧ 0x3);
bitcount ← bytecount × 8;
shift ← (address ∧ 0x3) × 8;
mem ← ZeroExtendbitcount(ReadMemoryLowbitcount(address));
IF (IsLittleEndian())
   result ← ZeroExtend32(mem);
ELSE
   result ← SignExtend32(mem << shift);
Rd ← Register(result);
```

Description:

This instruction loads the low part of a misaligned long-word from memory into R_d. The effective address is formed by adding the sign-extended 6-bit immediate s to R_m. The effective address points to the lowest byte in the misaligned long-word. The address of the highest byte in the low part of the misaligned long-word is determined by setting the least significant 2 bits of the effective address to 1.

This instruction loads the inclusive range of memory bytes starting at that lowest byte and ending at that highest byte. If the effective address is actually 4-byte aligned, then all 4 bytes are loaded. The loaded bytes are placed into the appropriate bytes within R_d, and other bytes are set to 0 or a sign-extension of bit 31 as required.

This instruction can be used in conjunction with LDHI.L and OR to load and sign-extend a misaligned long-word into a register. In this case, the LDHI.L effective address should be 3 bytes larger than the LDLO.L effective address.

Possible exceptions:

RADDERR, RTLBMISS, READPROT

LDLO.L Byte Mappings:

The mapping between byte locations in memory and byte positions in the destination register is shown below for each value of the low 2 bits in the effective address (EA). Each byte in the register is 0, a sign-extension or maps to the given memory address.

Little endian mode

Low 2 bits of EA↓	Bit 63 Byte 7	Byte 6	Byte 5	Byte 4	Byte 3	Byte 2	Byte 1	Byte 0 Bit 0
	Target register							
0x0	0	0	0	0	EA+3	EA+2	EA+1	EA+0
0x1	0	0	0	0	0	EA+2	EA+1	EA+0
0x2	0	0	0	0	0	0	EA+1	EA+0
0x3	0	0	0	0	0	0	0	EA+0

Big endian mode

Low 2 bits of EA↓	Bit 63 Byte 0	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7 Bit 0
	Target register							
0x0	sign extension of bit 31				EA+0	EA+1	EA+2	EA+3
0x1	sign extension of bit 31				EA+0	EA+1	EA+2	0
0x2	sign extension of bit 31				EA+0	EA+1	0	0
0x3	sign extension of bit 31				EA+0	0	0	0

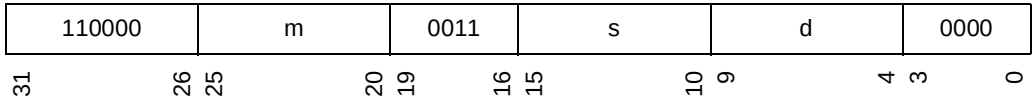
Notes:

The ‘disp’ in the assembly syntax represents the immediate s after sign extension.

When the memory access for LDLO.L causes an exception, the TEA control register is initialized with the effective address of the access. This corresponds to the address of the lowest byte in the misaligned long-word.

LDLO.Q Rm, disp, Rd

LDLO.Q Rm, disp, Rd




```
base ← ZeroExtend64(Rm);
disp ← SignExtend6(s);
address ← ZeroExtend64(base + disp);
bytecount ← 8 - (address ∧ 0x7);
bitcount ← bytecount × 8;
shift ← (address ∧ 0x7) × 8;
mem ← ZeroExtendbitcount(ReadMemoryLowbitcount(address));
IF (IsLittleEndian())
   result ← ZeroExtend64(mem);
ELSE
   result ← ZeroExtend64(mem << shift);
Rd ← Register(result);
```

Description:

This instruction loads the low part of a misaligned quad-word from memory into R_d. The effective address is formed by adding the sign-extended 6-bit immediate s to R_m. The effective address points to the lowest byte in the misaligned quad-word. The address of the highest byte in the low part of the misaligned quad-word is determined by setting the least significant 3 bits of the effective address to 1.

This instruction loads the inclusive range of memory bytes starting at that lowest byte and ending at that highest byte. If the effective address is actually 8-byte aligned, then all 8 bytes are loaded. The loaded bytes are placed into the appropriate bytes within R_d, and any other bytes are set to 0.

This instruction can be used in conjunction with LDHI.Q and OR to load a misaligned quad-word from memory into a register. In this case, the LDHI.Q effective address should be 7 bytes larger than the LDLO.Q effective address.

Possible exceptions:

RADDERR, RTLBMISS, READPROT

LDLO.Q Byte Mappings:

The mapping between byte locations in memory and byte positions in the destination register is shown below for each value of the low 3 bits in the effective address (EA). Each byte in the register is 0 or maps to the given memory address.

Little endian mode

Low 3 bits of EA↓	Bit 63 Target register Bit 0							
	Byte 7	**Byte 6**	**Byte 5**	**Byte 4**	**Byte 3**	**Byte 2**	**Byte 1**	**Byte 0**
0x0	EA+7	EA+6	EA+5	EA+4	EA+3	EA+2	EA+1	EA+0
0x1	0	EA+6	EA+5	EA+4	EA+3	EA+2	EA+1	EA+0
0x2	0	0	EA+5	EA+4	EA+3	EA+2	EA+1	EA+0
0x3	0	0	0	EA+4	EA+3	EA+2	EA+1	EA+0
0x4	0	0	0	0	EA+3	EA+2	EA+1	EA+0
0x5	0	0	0	0	0	EA+2	EA+1	EA+0
0x6	0	0	0	0	0	0	EA+1	EA+0
0x7	0	0	0	0	0	0	0	EA+0

Big endian mode

Low 3 bits of EA↓	Bit 63 Target register Bit 0							
	Byte 0	**Byte 1**	**Byte 2**	**Byte 3**	**Byte 4**	**Byte 5**	**Byte 6**	**Byte 7**
0x0	EA+0	EA+1	EA+2	EA+3	EA+4	EA+5	EA+6	EA+7
0x1	EA+0	EA+1	EA+2	EA+3	EA+4	EA+5	EA+6	0
0x2	EA+0	EA+1	EA+2	EA+3	EA+4	EA+5	0	0
0x3	EA+0	EA+1	EA+2	EA+3	EA+4	0	0	0
0x4	EA+0	EA+1	EA+2	EA+3	0	0	0	0

Big endian mode	Bit 63	Target register						Bit 0
Low 3 bits of EA↓	Byte 0	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7
0x5	EA+0	EA+1	EA+2	0	0	0	0	0
0x6	EA+0	EA+1	0	0	0	0	0	0
0x7	EA+0	0	0	0	0	0	0	0

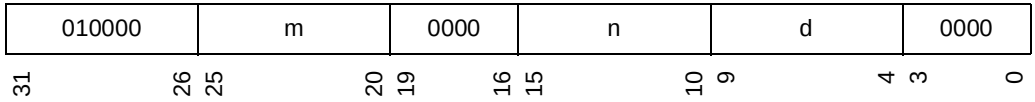
Notes:

The ‘disp’ in the assembly syntax represents the immediate s after sign extension.

When the memory access for LDLO.Q causes an exception, the TEA control register is initialized with the effective address of the access. This corresponds to the address of the lowest byte in the misaligned quad-word.

LDX.B Rm, Rn, Rd

LDX.B Rm, Rn, Rd




```
d_field ← ZeroExtend6(d);
base ← ZeroExtend64(Rm);
index ← SignExtend64(Rn);
address ← ZeroExtend64(base + index);
IF (d_field = 63)
   result ← PrefetchMemory(address);
ELSE
   result ← SignExtend8(ReadMemory8(address));
Rd ← Register(result);
```

Description:

This instruction loads a byte from the effective address formed by adding R_m and R_n. If the destination register is not $R_{63,}$ the loaded byte is sign-extended and placed in R_d. In exceptional cases, an appropriate exception is raised.

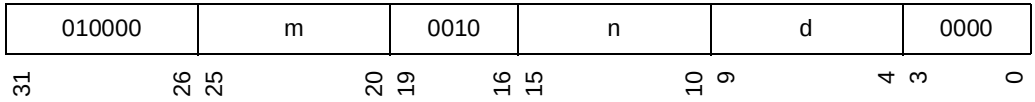
If the destination register is R_{63}, this indicates a software-directed data prefetch from the specified effective address. Software can use this instruction to give advance notice that particular data will be required. It is implementation-specific as to whether a prefetch will be performed. The effective address identifies a surrounding block of memory, which starts at an address aligned to the cache block size and has a size equal to the cache block size. The cache block size is implementation dependent. In exceptional cases, no exception is raised and the prefetch has no effect.

Possible exceptions:

RADDERR, RTLBMISS, READPROT

LDX.L Rm, Rn, Rd

LDX.L Rm, Rn, Rd




```
d_field ← ZeroExtend6(d);
base ← ZeroExtend64(Rm);
index ← SignExtend64(Rn);
address ← ZeroExtend64(base + index);
IF (d_field = 63)
   result ← PrefetchMemory(address);
ELSE
   result ← SignExtend32(ReadMemory32(address));
Rd ← Register(result);
```

Description:

This instruction loads a long-word from the effective address formed by adding R_m and R_n. If the destination register is not R_{63}, the loaded long-word is sign-extended and placed in R_d. Note that only one version of this instruction is provided: the representation of signed and unsigned 32-bit data in a register is the same. In exceptional cases, including misaligned loads, an appropriate exception is raised.

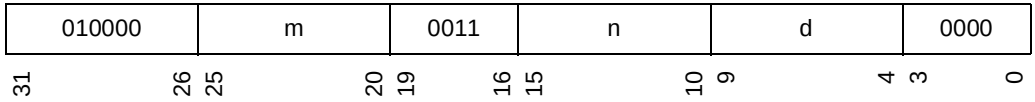
If the destination register is R_{63}, this indicates a software-directed data prefetch from the specified effective address. Software can use this instruction to give advance notice that particular data will be required. It is implementation-specific as to whether a prefetch will be performed. The effective address identifies a surrounding block of memory, which starts at an address aligned to the cache block size and has a size equal to the cache block size. The cache block size is implementation dependent. In exceptional cases, no exception is raised and the prefetch has no effect.

Possible exceptions:

RADDERR, RTLBMISS, READPROT

LDX.Q Rm, Rn, Rd

LDX.Q Rm, Rn, Rd




```
d_field ← ZeroExtend6(d);
base ← ZeroExtend64(Rm);
index ← SignExtend64(Rn);
address ← ZeroExtend64(base + index);
IF (d_field = 63)
   result ← PrefetchMemory(address);
ELSE
   result ← ZeroExtend64(ReadMemory64(address));
Rd ← Register(result);
```

Description:

This instruction loads a quad-word from the effective address formed by adding R_m and R_n. If the destination register is not R_{63}, the loaded quad-word is placed in R_d. Note that only one version of this instruction is provided: sign is unimportant when loading an object of the same size as a register. In exceptional cases, including misaligned loads, an appropriate exception is raised.

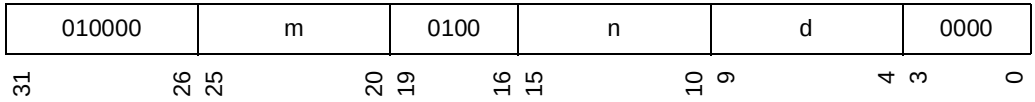
If the destination register is R_{63}, this indicates a software-directed data prefetch from the specified effective address. Software can use this instruction to give advance notice that particular data will be required. It is implementation-specific as to whether a prefetch will be performed. The effective address identifies a surrounding block of memory, which starts at an address aligned to the cache block size and has a size equal to the cache block size. The cache block size is implementation dependent. In exceptional cases, no exception is raised and the prefetch has no effect.

Possible exceptions:

RADDERR, RTLBMISS, READPROT

LDX.UB Rm, Rn, Rd

LDX.UB Rm, Rn, Rd




```
d_field ← ZeroExtend6(d);
base ← ZeroExtend64(Rm);
index ← SignExtend64(Rn);
address ← ZeroExtend64(base + index);
IF (d_field = 63)
   result ← PrefetchMemory(address);
ELSE
   result ← ZeroExtend8(ReadMemory8(address));
Rd ← Register(result);
```

Description:

This instruction loads a byte from the effective address formed by adding R_m and R_n. If the destination register is not $R_{63,}$ the loaded byte is zero-extended and placed in R_d. In exceptional cases, an appropriate exception is raised.

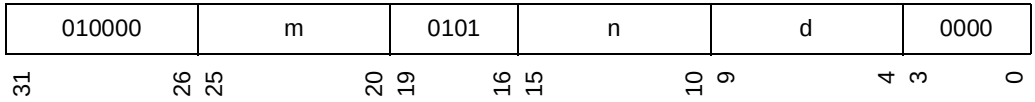
If the destination register is R_{63}, this indicates a software-directed data prefetch from the specified effective address. Software can use this instruction to give advance notice that particular data will be required. It is implementation-specific as to whether a prefetch will be performed. The effective address identifies a surrounding block of memory, which starts at an address aligned to the cache block size and has a size equal to the cache block size. The cache block size is implementation dependent. In exceptional cases, no exception is raised and the prefetch has no effect.

Possible exceptions:

RADDERR, RTLBMISS, READPROT

LDX.UW Rm, Rn, Rd

LDX.UW Rm, Rn, Rd




```
d_field ← ZeroExtend6(d);
base ← ZeroExtend64(Rm);
index ← SignExtend64(Rn);
address ← ZeroExtend64(base + index);
IF (d_field = 63)
   result ← PrefetchMemory(address);
ELSE
   result ← ZeroExtend16(ReadMemory16(address));
Rd ← Register(result);
```

Description:

This instruction loads a word from the effective address formed by adding R_m and R_n. If the destination register is not R_{63}, the loaded word is zero-extended and placed in R_d. In exceptional cases, including misaligned loads, an appropriate exception is raised.

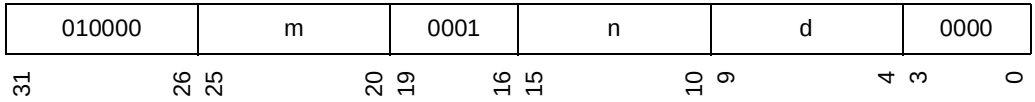
If the destination register is R_{63}, this indicates a software-directed data prefetch from the specified effective address. Software can use this instruction to give advance notice that particular data will be required. It is implementation-specific as to whether a prefetch will be performed. The effective address identifies a surrounding block of memory, which starts at an address aligned to the cache block size and has a size equal to the cache block size. The cache block size is implementation dependent. In exceptional cases, no exception is raised and the prefetch has no effect.

Possible exceptions:

RADDERR, RTLBMISS, READPROT

LDX.W Rm, Rn, Rd

LDX.W Rm, Rn, Rd




```
d_field ← ZeroExtend6(d);
base ← ZeroExtend64(Rm);
index ← SignExtend64(Rn);
address ← ZeroExtend64(base + index);
IF (d_field = 63)
   result ← PrefetchMemory(address);
ELSE
   result ← SignExtend16(ReadMemory16(address));
Rd ← Register(result);
```

Description:

This instruction loads a word from the effective address formed by adding R_m and R_n. If the destination register is not R_{63}, the loaded word is sign-extended and placed in R_d. In exceptional cases, including misaligned loads, an appropriate exception is raised.

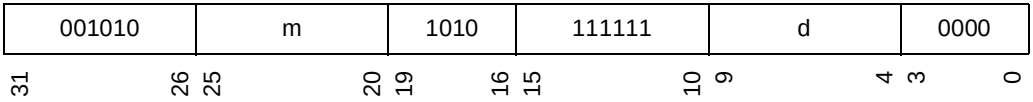
If the destination register is R_{63}, this indicates a software-directed data prefetch from the specified effective address. Software can use this instruction to give advance notice that particular data will be required. It is implementation-specific as to whether a prefetch will be performed. The effective address identifies a surrounding block of memory, which starts at an address aligned to the cache block size and has a size equal to the cache block size. The cache block size is implementation dependent. In exceptional cases, no exception is raised and the prefetch has no effect.

Possible exceptions:

RADDERR, RTLBMISS, READPROT

MABS.L Rm, Rd

MABS.L Rm, Rd

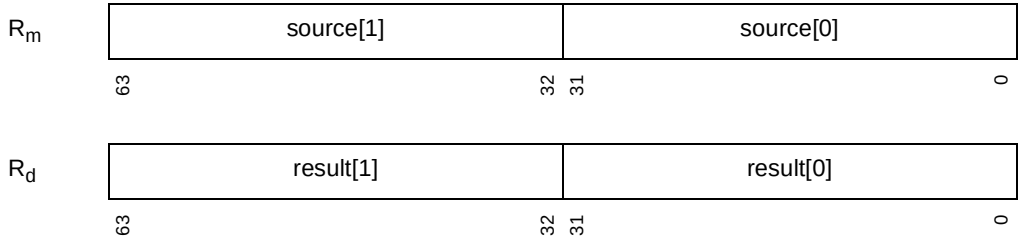



```
source ← MultiSignExtend32(Rm);
REPEAT i FROM 0 FOR 2
   IF (source[i] ≥ 0)
      result[i] ← source[i];
   ELSE
      result[i] ← SignedSaturate32(- source[i]);
Rd ← MultiRegister32(result);
```

Description:

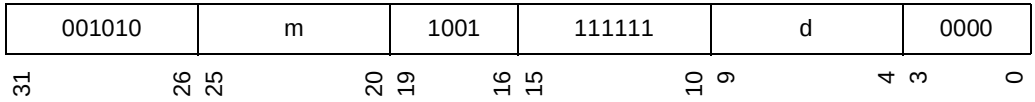
This multimedia instruction calculates the absolute value of each of the packed 32-bit elements held in R_m and places the packed results in R_d.

Multimedia Formats:



MABS.W Rm, Rd

MABS.W Rm, Rd

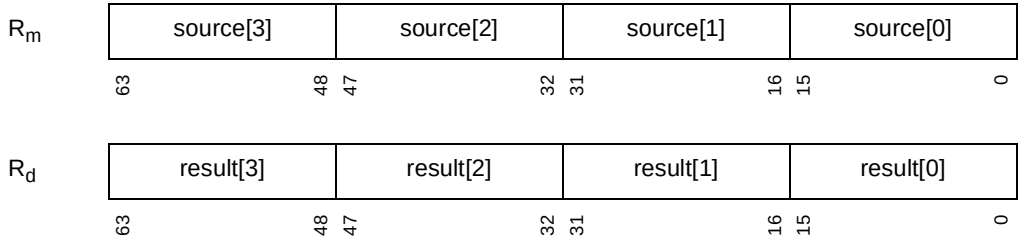



```
source ← MultiSignExtend16(Rm);
REPEAT i FROM 0 FOR 4
   IF (source[i] ≥ 0)
      result[i] ← source[i];
   ELSE
      result[i] ← SignedSaturate16(- source[i]);
Rd ← MultiRegister16(result);
```

Description:

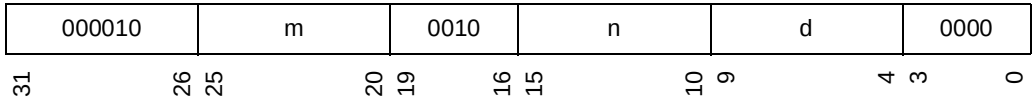
This multimedia instruction calculates the absolute value of each of the packed 16-bit elements held in R_m and places the packed results in R_d.

Multimedia Formats:



MADD.L Rm, Rn, Rd

MADD.L Rm, Rn, Rd

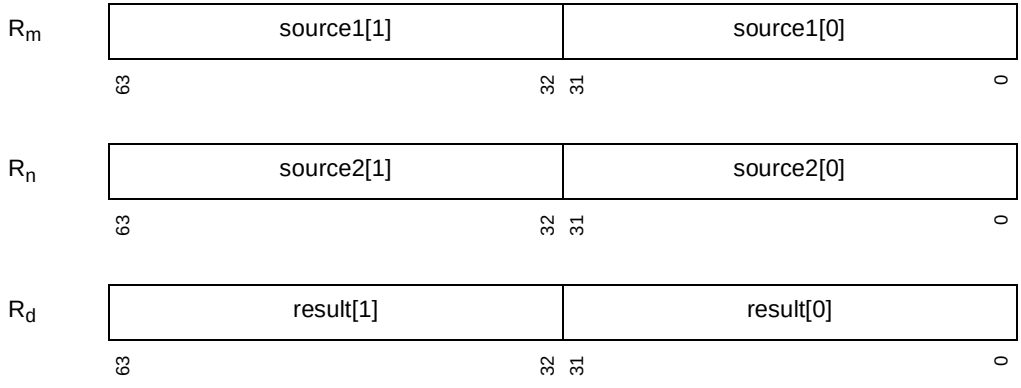



```
source1 ← MultiZeroExtend32(Rm);
source2 ← MultiZeroExtend32(Rn);
REPEAT i FROM 0 FOR 2
   result[i] ← ZeroExtend32(source1[i] + source2[i]);
Rd ← MultiRegister32(result);
```

Description:

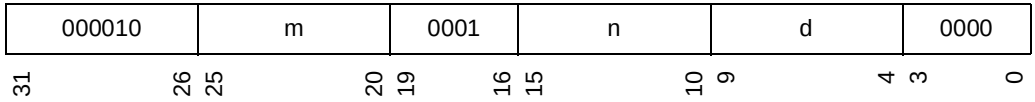
This multimedia instruction performs modulo, 32-bit addition on corresponding packed 32-bit elements held in R_m and R_n, and places the packed results in R_d. Sign is unimportant for modulo arithmetic and so this instruction can be used on both signed and unsigned types.

Multimedia Formats:



MADD.W Rm, Rn, Rd

MADD.W Rm, Rn, Rd

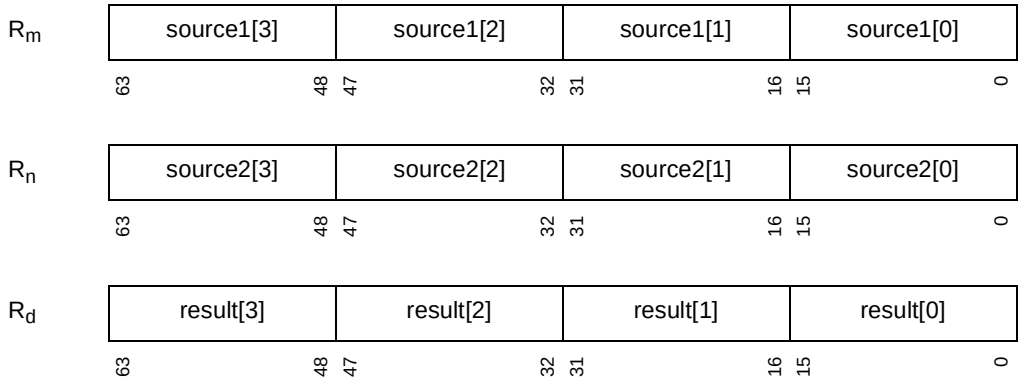



```
source1 ← MultiZeroExtend16(Rm);
source2 ← MultiZeroExtend16(Rn);
REPEAT i FROM 0 FOR 4
   result[i] ← ZeroExtend16(source1[i] + source2[i]);
Rd ← MultiRegister16(result);
```

Description:

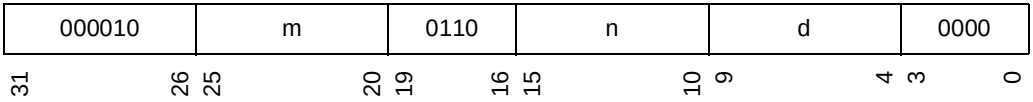
This multimedia instruction performs modulo, 16-bit addition on corresponding packed 16-bit elements held in R_m and R_n, and places the packed results in R_d. Sign is unimportant for modulo arithmetic and so this instruction can be used on both signed and unsigned types.

Multimedia Formats:



MADDS.L Rm, Rn, Rd

MADDS.L Rm, Rn, Rd

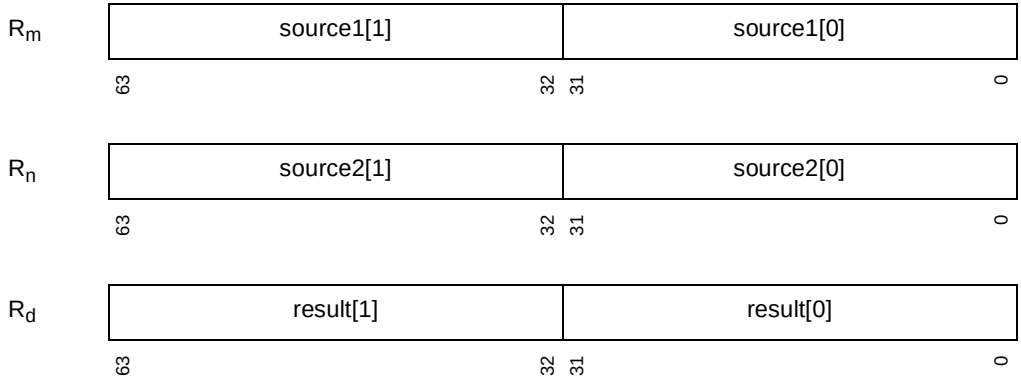



```
source1 ← MultiSignExtend32(Rm);
source2 ← MultiSignExtend32(Rn);
REPEAT i FROM 0 FOR 2
   result[i] ← SignedSaturate32(source1[i] + source2[i]);
Rd ← MultiRegister32(result);
```

Description:

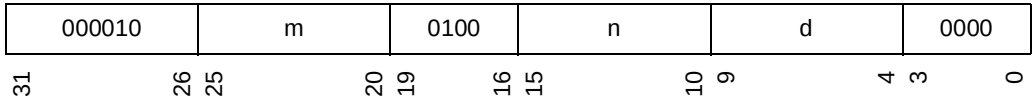
This multimedia instruction performs saturating, signed, 32-bit addition on corresponding packed 32-bit elements held in R_m and R_n, and places the packed results in R_d. The additions are saturated to the signed range $[-2^{31}, 2^{31}]$.

Multimedia Formats:



MADDS.UB Rm, Rn, Rd

MADDS.UB Rm, Rn, Rd

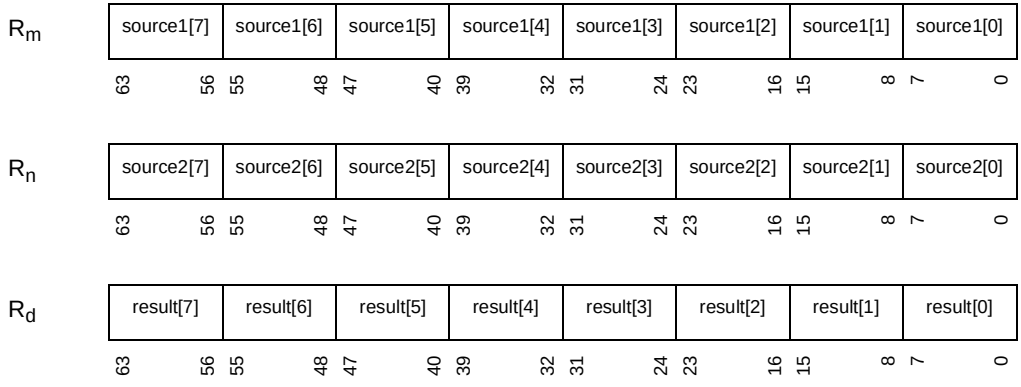



```
source1 ← MultiZeroExtend8(Rm);
source2 ← MultiZeroExtend8(Rn);
REPEAT i FROM 0 FOR 8
   result[i] ← UnsignedSaturate8(source1[i] + source2[i]);
Rd ← MultiRegister8(result);
```

Description:

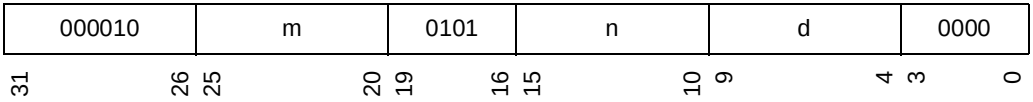
This multimedia instruction performs saturating, unsigned, 8-bit addition on corresponding packed 8-bit elements held in R_m and R_n, and places the packed results in R_d. The additions are saturated to the unsigned range [0, 256].

Multimedia Formats:



MADDS.W Rm, Rn, Rd

MADDS.W Rm, Rn, Rd

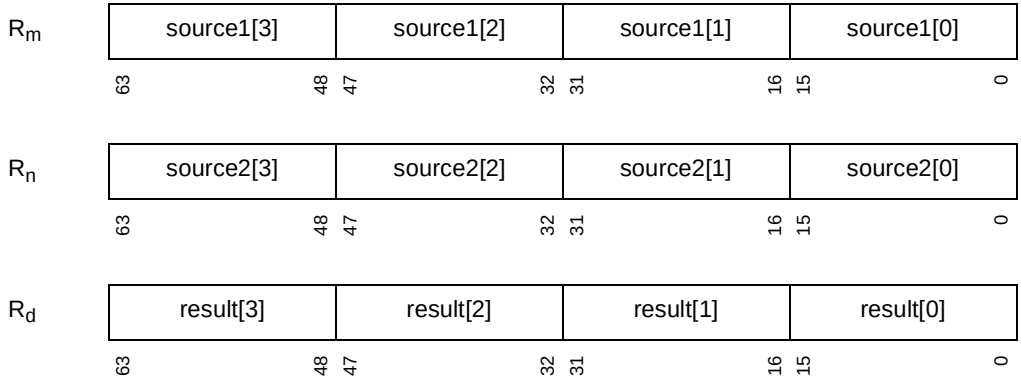



```
source1 ← MultiSignExtend16(Rm);
source2 ← MultiSignExtend16(Rn);
REPEAT i FROM 0 FOR 4
   result[i] ← SignedSaturate16(source1[i] + source2[i]);
Rd ← MultiRegister16(result);
```

Description:

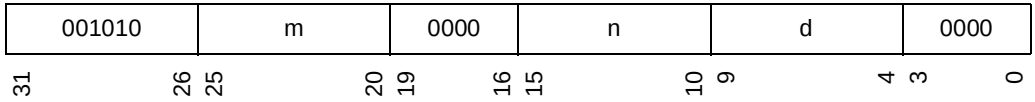
This multimedia instruction performs saturating, signed, 16-bit addition on corresponding packed 16-bit elements held in R_m and R_n, and places the packed results in R_d. The additions are saturated to the signed range $[-2^{15}, 2^{15}]$.

Multimedia Formats:



MCMPEQ.B Rm, Rn, Rd

MCMPEQ.B Rm, Rn, Rd

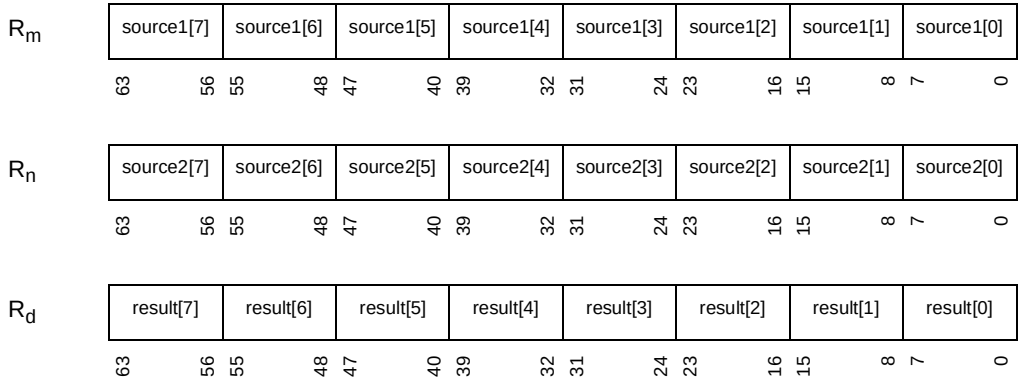



```
source1 ← MultiSignExtend8(Rm);
source2 ← MultiSignExtend8(Rn);
REPEAT i FROM 0 FOR 8
   IF (source1[i] = source2[i])
      result[i] ← 0xff;
   ELSE
      result[i] ← 0x00;
Rd ← MultiRegister8(result);
```

Description:

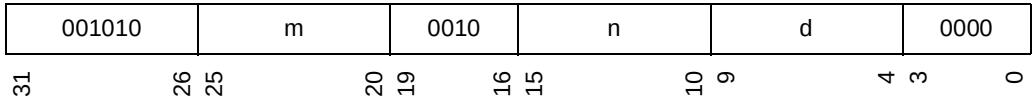
This multimedia instruction compares corresponding packed 8-bit elements held in R_m and R_n for equality, and places the packed boolean results in R_d. Boolean false is represented by an all-zeroes element and boolean true by an all-ones element.

Multimedia Formats:



MCMPEQ.L Rm, Rn, Rd

MCMPEQ.L Rm, Rn, Rd

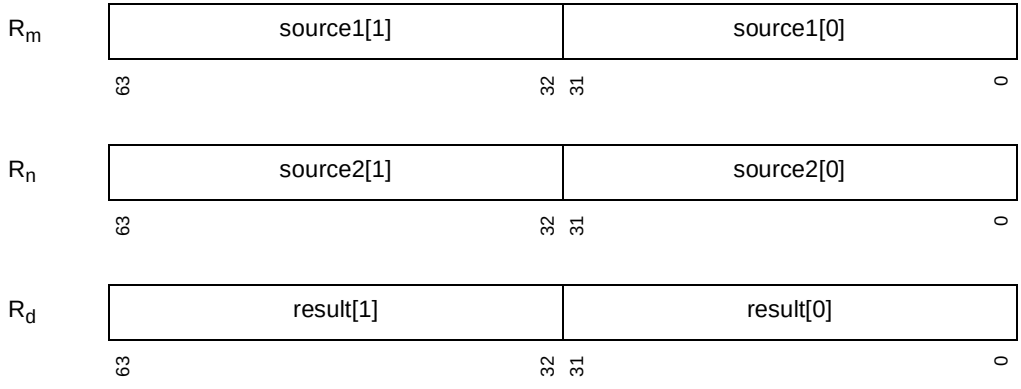



```
source1 ← MultiSignExtend32(Rm);
source2 ← MultiSignExtend32(Rn);
REPEAT i FROM 0 FOR 2
   IF (source1[i] = source2[i])
      result[i] ← 0xffffffff;
   ELSE
      result[i] ← 0x00000000;
Rd ← MultiRegister32(result);
```

Description:

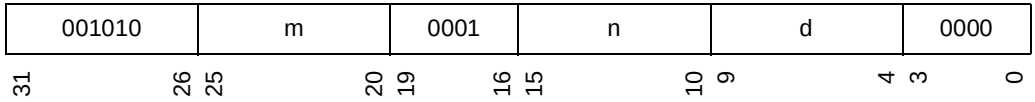
This multimedia instruction compares corresponding packed 32-bit elements held in R_m and R_n for equality, and places the packed boolean results in R_d. Boolean false is represented by an all-zeroes element and boolean true by an all-ones element.

Multimedia Formats:



MCMPEQ.W Rm, Rn, Rd

MCMPEQ.W Rm, Rn, Rd

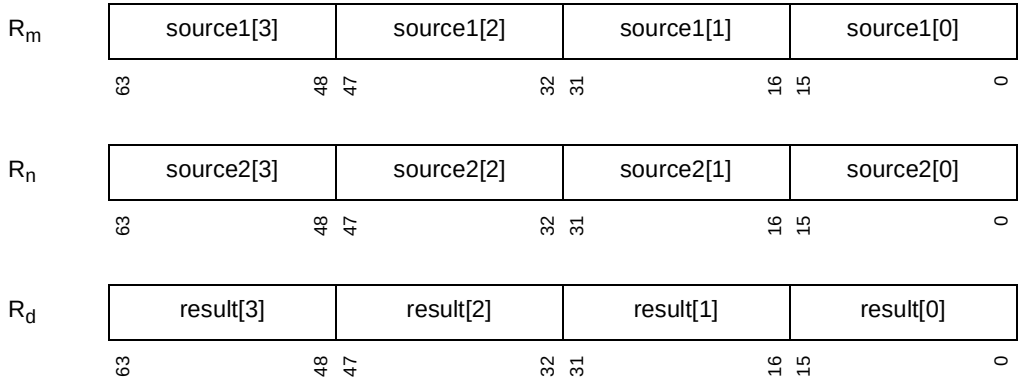



```
source1 ← MultiSignExtend16(Rm);
source2 ← MultiSignExtend16(Rn);
REPEAT i FROM 0 FOR 4
   IF (source1[i] = source2[i])
      result[i] ← 0xffff;
   ELSE
      result[i] ← 0x0000;
Rd ← MultiRegister16(result);
```

Description:

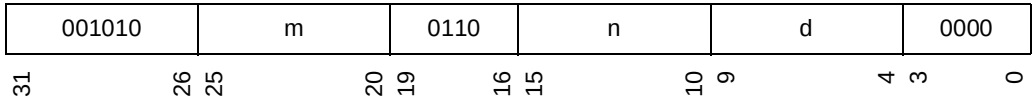
This multimedia instruction compares corresponding packed 16-bit elements held in R_m and R_n for equality, and places the packed boolean results in R_d. Boolean false is represented by an all-zeroes element and boolean true by an all-ones element.

Multimedia Formats:



MCMPGT.L Rm, Rn, Rd

MCMPGT.L Rm, Rn, Rd

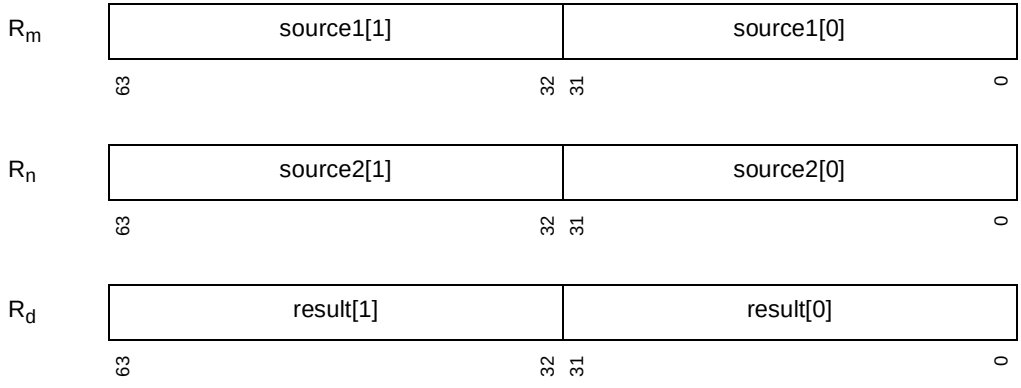



```
source1 ← MultiSignExtend32(Rm);
source2 ← MultiSignExtend32(Rn);
REPEAT i FROM 0 FOR 2
   IF (source1[i] > source2[i])
      result[i] ← 0xffffffff;
   ELSE
      result[i] ← 0x00000000;
Rd ← MultiRegister32(result);
```

Description:

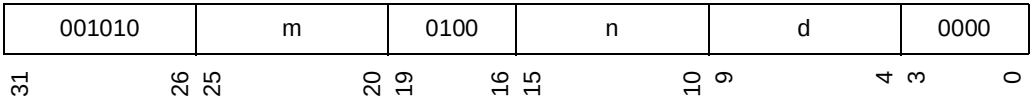
This multimedia instruction compares corresponding packed 32-bit elements held in R_m and R_n to test whether each signed R_m element is greater than the corresponding signed R_n element, and places the packed boolean results in R_d. Boolean false is represented by an all-zeroes element and boolean true by an all-ones element.

Multimedia Formats:



MCMPGT.UB Rm, Rn, Rd

MCMPGT.UB Rm, Rn, Rd

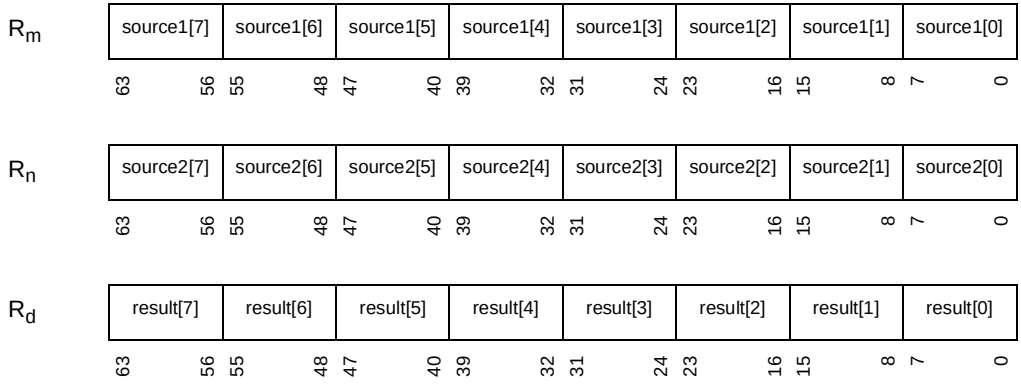



```
source1 ← MultiZeroExtend8(Rm);
source2 ← MultiZeroExtend8(Rn);
REPEAT i FROM 0 FOR 8
   IF (source1[i] > source2[i])
      result[i] ← 0xff;
   ELSE
      result[i] ← 0x00;
Rd ← MultiRegister8(result);
```

Description:

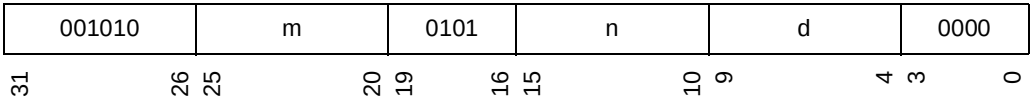
This multimedia instruction compares corresponding packed 8-bit elements held in R_m and R_n to test whether each unsigned R_m element is greater than the corresponding unsigned R_n element, and places the packed boolean results in R_d. Boolean false is represented by an all-zeroes element and boolean true by an all-ones element.

Multimedia Formats:



MCMPGT.W Rm, Rn, Rd

MCMPGT.W Rm, Rn, Rd

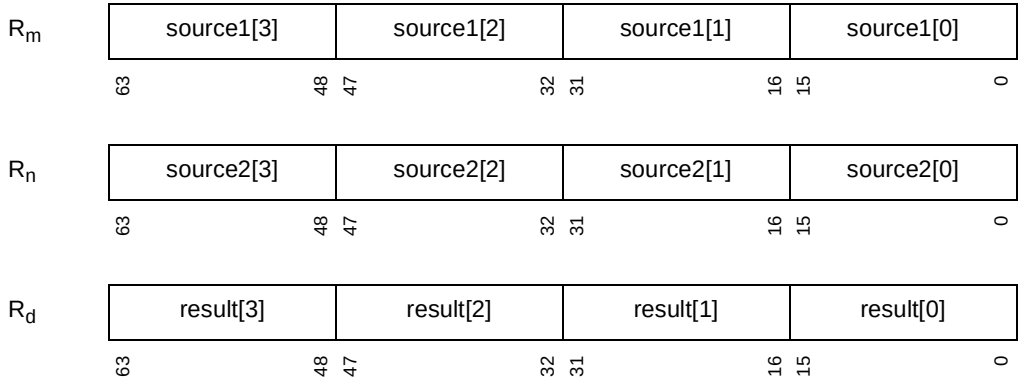



```
source1 ← MultiSignExtend16(Rm);
source2 ← MultiSignExtend16(Rn);
REPEAT i FROM 0 FOR 4
   IF (source1[i] > source2[i])
      result[i] ← 0xffff;
   ELSE
      result[i] ← 0x0000;
Rd ← MultiRegister16(result);
```

Description:

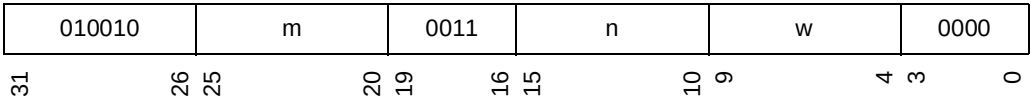
This multimedia instruction compares corresponding packed 16-bit elements held in R_m and R_n to test whether each signed R_m element is greater than the corresponding signed R_n element, and places the packed boolean results in R_d. Boolean false is represented by an all-zeroes element and boolean true by an all-ones element.

Multimedia Formats:



MCMV Rm, Rn, Rw

MCMV Rm, Rn, Rw

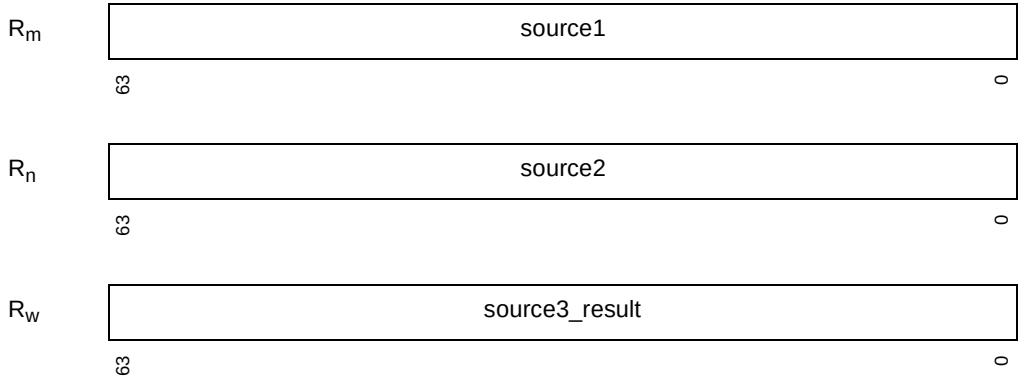



```
source1 ← ZeroExtend64(Rm);
source2 ← ZeroExtend64(Rn);
source3_result ← ZeroExtend64(Rw);
source3_result ← (source1 ∧ source2) ∨ (source3_result ∧ (~ source2));
Rw ← Register(source3_result);
```

Description:

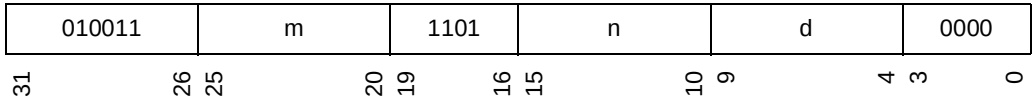
This multimedia instruction performs a bitwise conditional move from R_m to R_w based on the value provided in the mask R_n. If bit i, where i is in [0,63], of R_n is 1 then bit i of R_m is copied to bit i of R_w, otherwise bit i of R_w is left unchanged.

Multimedia Formats:



MCNVS.LW Rm, Rn, Rd

MCNVS.LW Rm, Rn, Rd

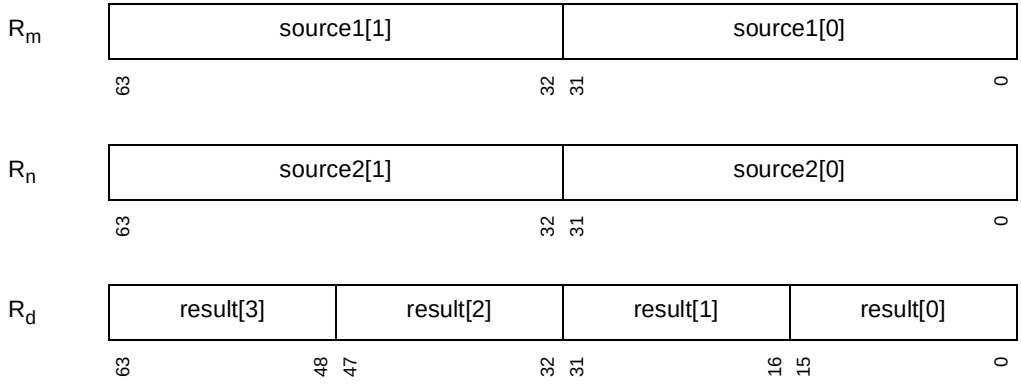



```
source1 ← MultiSignExtend32(Rm);
source2 ← MultiSignExtend32(Rn);
REPEAT i FROM 0 FOR 2
   result[i] ← SignedSaturate16(source1[i]);
REPEAT i FROM 0 FOR 2
   result[i + 2] ← SignedSaturate16(source2[i]);
Rd ← MultiRegister16(result);
```

Description:

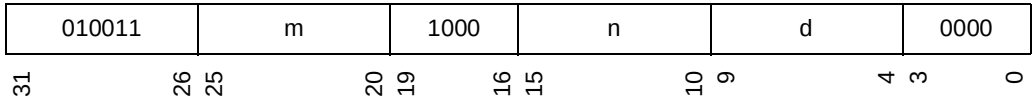
This multimedia instruction performs saturating, down-conversions from the packed 32-bit elements held in R_m and R_n to signed 16-bit values, and places the packed results in R_d. The results from the R_m conversions are placed in the lower half of R_d, and those from the R_n conversions in the upper half of R_d. The values are saturated to the signed range $[-2^{15}, 2^{15}]$.

Multimedia Formats:



MCNVS.WB Rm, Rn, Rd

MCNVS.WB Rm, Rn, Rd

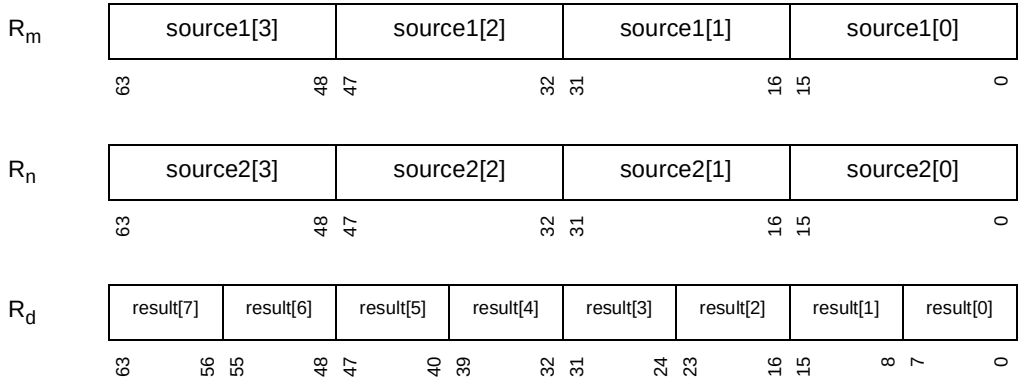



```
source1 ← MultiSignExtend16(Rm);
source2 ← MultiSignExtend16(Rn);
REPEAT i FROM 0 FOR 4
   result[i] ← SignedSaturate8(source1[i]);
REPEAT i FROM 0 FOR 4
   result[i + 4] ← SignedSaturate8(source2[i]);
Rd ← MultiRegister8(result);
```

Description:

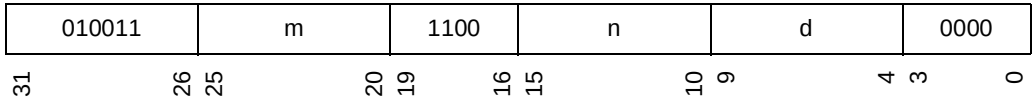
This multimedia instruction performs saturating, down-conversions from the packed 16-bit elements held in R_m and R_n to signed 8-bit values, and places the packed results in R_d. The results from the R_m conversions are placed in the lower half of R_d, and those from the R_n conversions in the upper half of R_d. The values are saturated to the signed range [-128, 128].

Multimedia Formats:



MCNVS.WUB Rm, Rn, Rd

MCNVS.WUB Rm, Rn, Rd

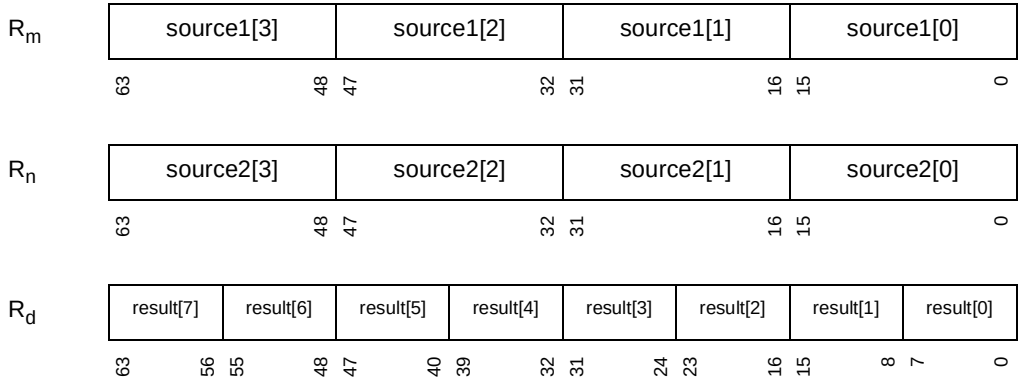



```
source1 ← MultiSignExtend16(Rm);
source2 ← MultiSignExtend16(Rn);
REPEAT i FROM 0 FOR 4
   result[i] ← UnsignedSaturate8(source1[i]);
REPEAT i FROM 0 FOR 4
   result[i + 4] ← UnsignedSaturate8(source2[i]);
Rd ← MultiRegister8(result);
```

Description:

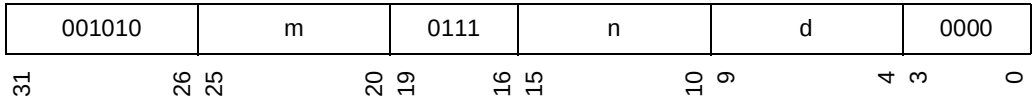
This multimedia instruction performs saturating, down-conversions from the packed 16-bit elements held in R_m and R_n to unsigned 8-bit values, and places the packed results in R_d. The results from the R_m conversions are placed in the lower half of R_d, and those from the R_n conversions in the upper half of R_d. The values are saturated to the unsigned range [0, 256].

Multimedia Formats:



MEXTR1 Rm, Rn, Rd

MEXTR1 Rm, Rn, Rd

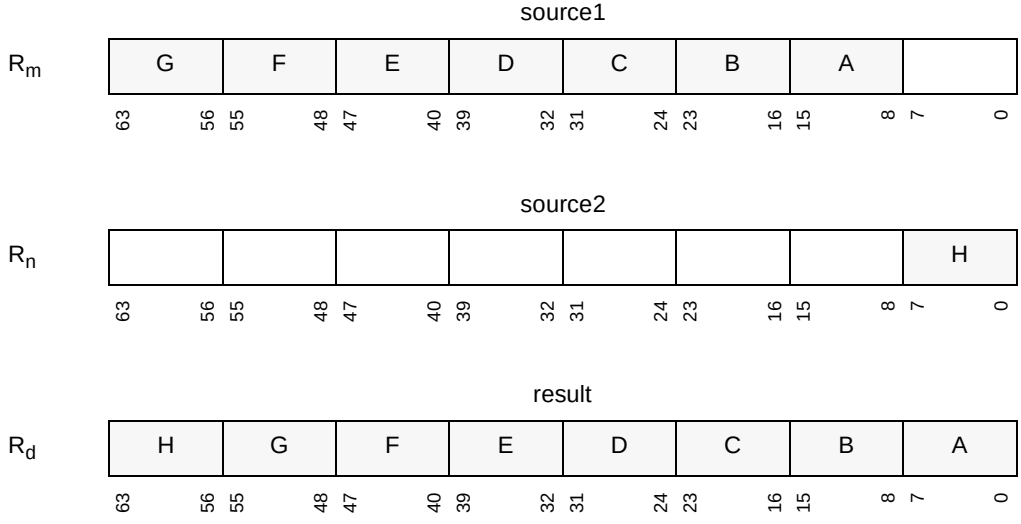



```
source1 ← ZeroExtend64(Rm);
source2 ← ZeroExtend64(Rn);
result ← (UpperBytes7(source1) >> (1 × 8)) ∨ (LowerBytes1(source2) << (7 × 8));
Rd ← Register(result);
```

Description:

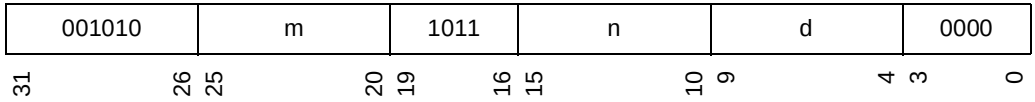
This multimedia instruction concatenates R_m and R_n together to form a 128-bit intermediate where the lower 64 bits are provided by R_m and the upper 64 bits by R_n. A 64-bit slice is extracted from this 128-bit intermediate starting at the $(1x8)^{th.}$ bit, and this result is placed in R_d.

Multimedia Formats:



MEXTR2 Rm, Rn, Rd

MEXTR2 Rm, Rn, Rd

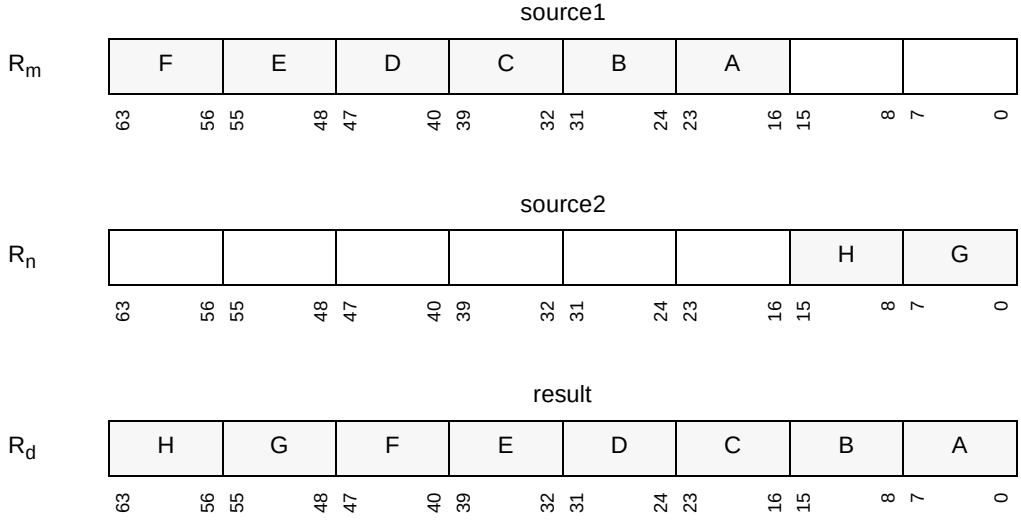



```
source1 ← ZeroExtend64(Rm);
source2 ← ZeroExtend64(Rn);
result ← (UpperBytes6(source1) >> (2 × 8)) ∨ (LowerBytes2(source2) << (6 × 8));
Rd ← Register(result);
```

Description:

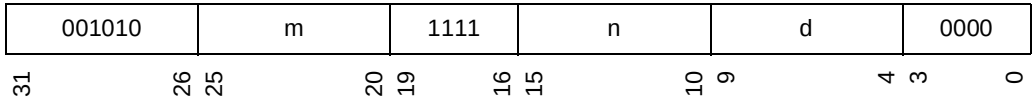
This multimedia instruction concatenates R_m and R_n together to form a 128-bit intermediate where the lower 64 bits are provided by R_m and the upper 64 bits by R_n. A 64-bit slice is extracted from this 128-bit intermediate starting at the $(2x8)^{th.}$ bit, and this result is placed in R_d.

Multimedia Formats:



MEXTR3 Rm, Rn, Rd

MEXTR3 Rm, Rn, Rd

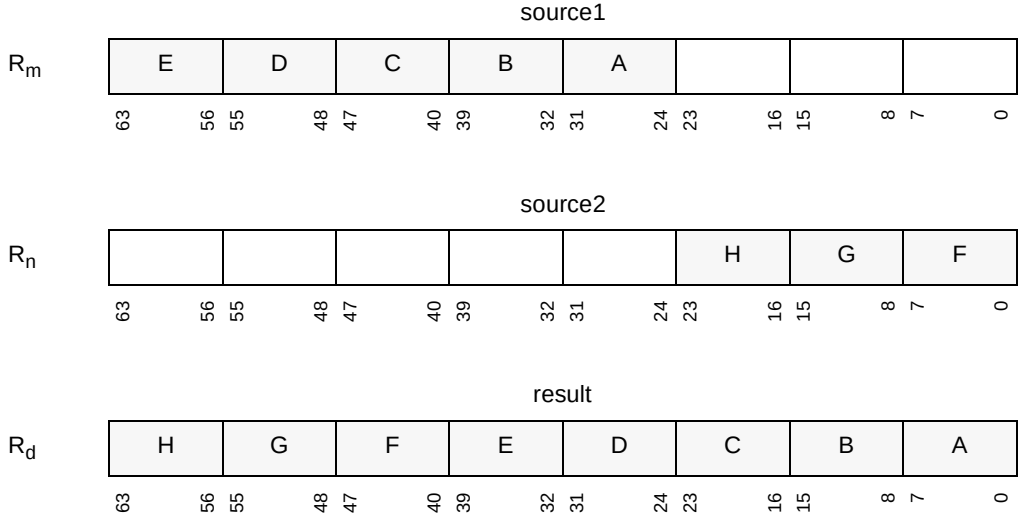


source1 ← ZeroExtend$_{64}$(R_m);
source2 ← ZeroExtend$_{64}$(R_n);
result ← (UpperBytes$_5$(source1) >> (3 × 8)) ∨ (LowerBytes$_3$(source2) << (5 × 8));
R_d ← Register(result);

Description:

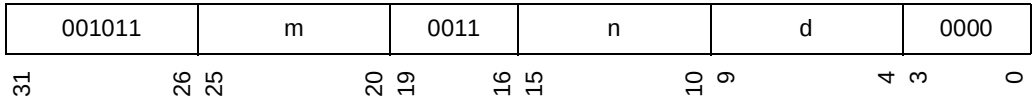
This multimedia instruction concatenates R_m and R_n together to form a 128-bit intermediate where the lower 64 bits are provided by R_m and the upper 64 bits by R_n. A 64-bit slice is extracted from this 128-bit intermediate starting at the $(3x8)^{th.}$ bit, and this result is placed in R_d.

Multimedia Formats:



MEXTR4 Rm, Rn, Rd

MEXTR4 Rm, Rn, Rd

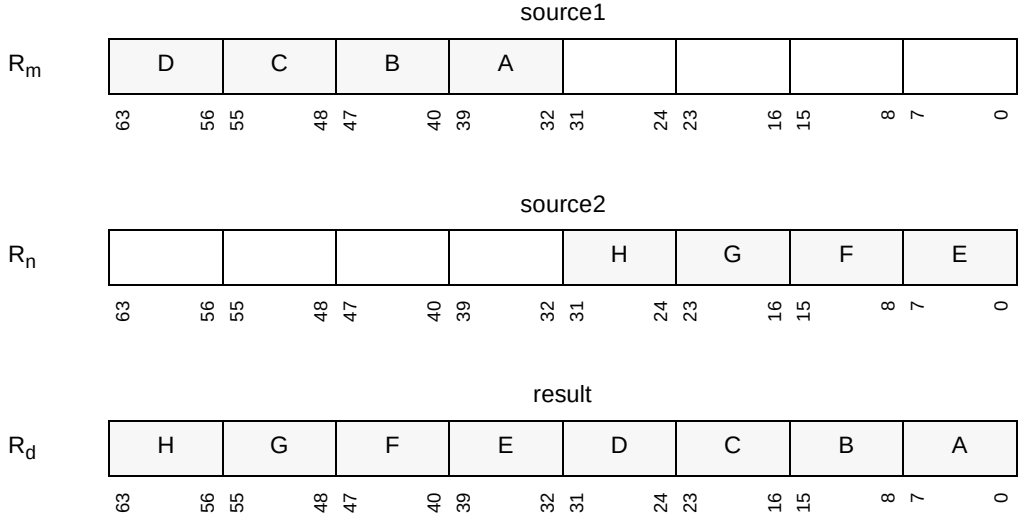


source1 ← ZeroExtend$_{64}$(R_m);
source2 ← ZeroExtend$_{64}$(R_n);
result ← (UpperBytes$_4$(source1) >> (4 × 8)) ∨ (LowerBytes$_4$(source2) << (4 × 8));
R_d ← Register(result);

Description:

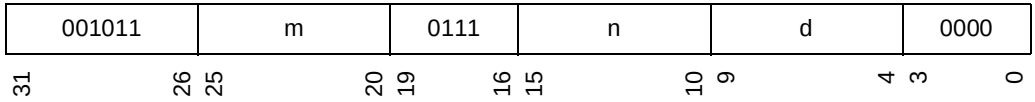
This multimedia instruction concatenates R_m and R_n together to form a 128-bit intermediate where the lower 64 bits are provided by R_m and the upper 64 bits by R_n. A 64-bit slice is extracted from this 128-bit intermediate starting at the $(4x8)^{th.}$ bit, and this result is placed in R_d.

Multimedia Formats:



MEXTR5 Rm, Rn, Rd

MEXTR5 Rm, Rn, Rd

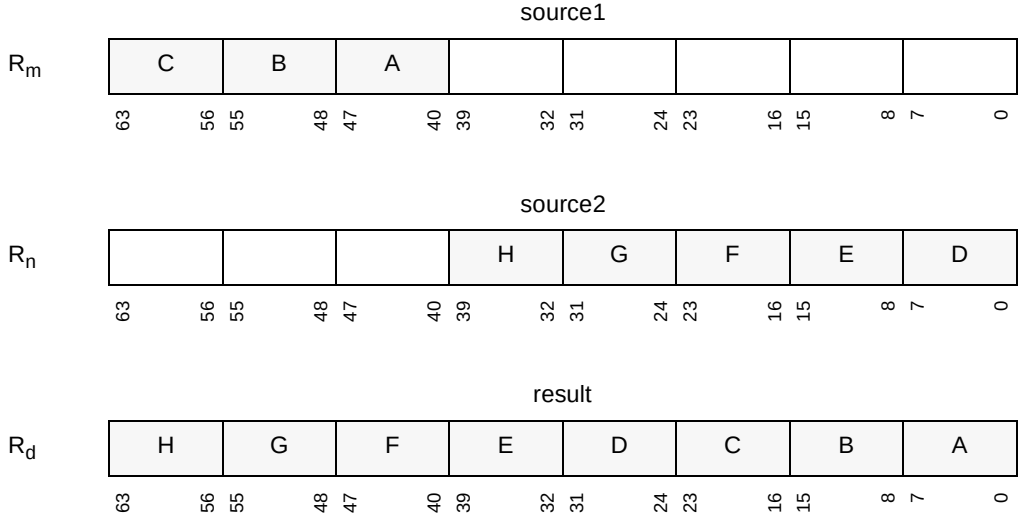


source1 ← ZeroExtend$_{64}$(R_m);
source2 ← ZeroExtend$_{64}$(R_n);
result ← (UpperBytes$_3$(source1) >> (5 × 8)) ∨ (LowerBytes$_5$(source2) << (3 × 8));
R_d ← Register(result);

Description:

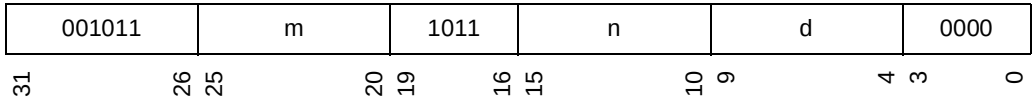
This multimedia instruction concatenates R_m and R_n together to form a 128-bit intermediate where the lower 64 bits are provided by R_m and the upper 64 bits by R_n. A 64-bit slice is extracted from this 128-bit intermediate starting at the $(5x8)^{th.}$ bit, and this result is placed in R_d.

Multimedia Formats:



MEXTR6 Rm, Rn, Rd

MEXTR6 Rm, Rn, Rd

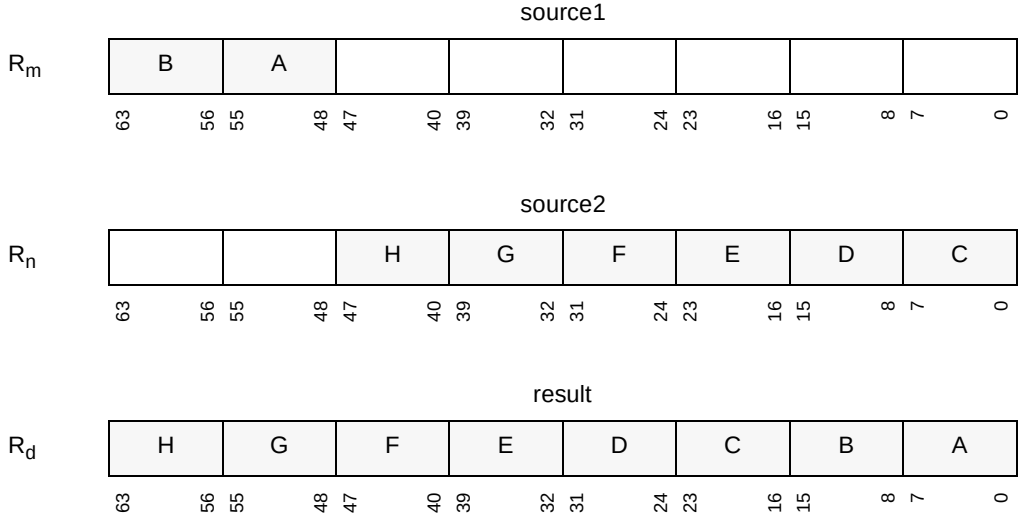


source1 ← ZeroExtend$_{64}$(R_m);
source2 ← ZeroExtend$_{64}$(R_n);
result ← (UpperBytes$_2$(source1) >> (6 × 8)) ∨ (LowerBytes$_6$(source2) << (2 × 8));
R_d ← Register(result);

Description:

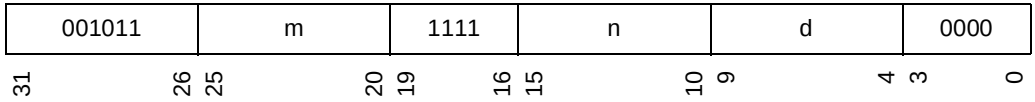
This multimedia instruction concatenates R_m and R_n together to form a 128-bit intermediate where the lower 64 bits are provided by R_m and the upper 64 bits by R_n. A 64-bit slice is extracted from this 128-bit intermediate starting at the (6x8)$^{th.}$ bit, and this result is placed in R_d.

Multimedia Formats:



MEXTR7 Rm, Rn, Rd

MEXTR7 Rm, Rn, Rd

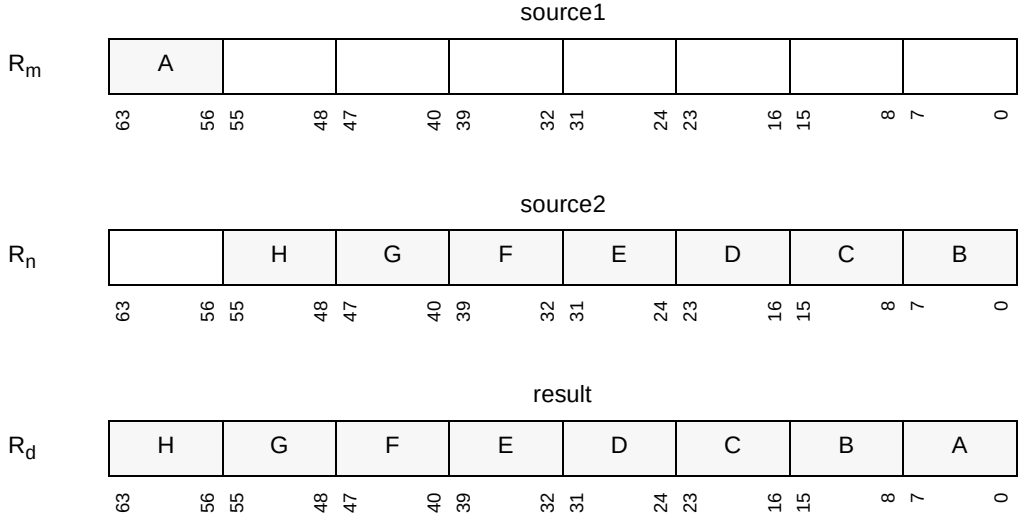


source1 ← ZeroExtend$_{64}$(R_m);
source2 ← ZeroExtend$_{64}$(R_n);
result ← (UpperBytes$_1$(source1) >> (7 × 8)) ∨ (LowerBytes$_7$(source2) << (1 × 8));
R_d ← Register(result);

Description:

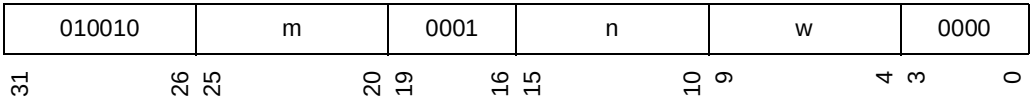
This multimedia instruction concatenates R_m and R_n together to form a 128-bit intermediate where the lower 64 bits are provided by R_m and the upper 64 bits by R_n. A 64-bit slice is extracted from this 128-bit intermediate starting at the $(7x8)^{th.}$ bit, and this result is placed in R_d.

Multimedia Formats:



MMACFX.WL Rm, Rn, Rw

MMACFX.WL Rm, Rn, Rw

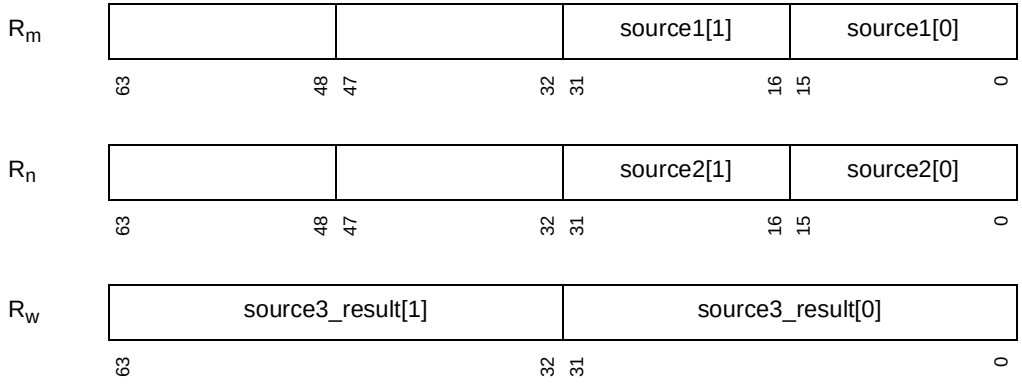



```
source1 ← MultiSignExtend16(Rm);
source2 ← MultiSignExtend16(Rn);
source3_result ← MultiSignExtend32(Rw);
REPEAT i FROM 0 FOR 2
{
   temp ← source1[i] × source2[i];
   temp ← SignedSaturate32(temp << 1);
   source3_result[i] ← SignedSaturate32(source3_result[i] + temp);
}
Rw ← MultiRegister32(source3_result);
```

Description:

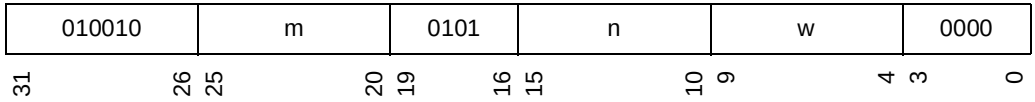
This multimedia instruction performs full-width, fractional multiplication on corresponding packed, signed 16-bit elements held in the lower halves of R_m and R_n, normalizes the results to a 32-bit fractional format, sums with corresponding packed, signed 32-bit elements held in R_w, saturates to the 32-bit signed fractional range, and places the packed 32-bit results in R_w. In the special case of a fractional multiply of -1 by -1, the multiplication result, which would otherwise be out of representable range, is saturated to the largest representable positive value before the summing and saturation stages. No rounding is necessary.

Multimedia Formats:



MMACNFX.WL Rm, Rn, Rw

MMACNFX.WL Rm, Rn, Rw

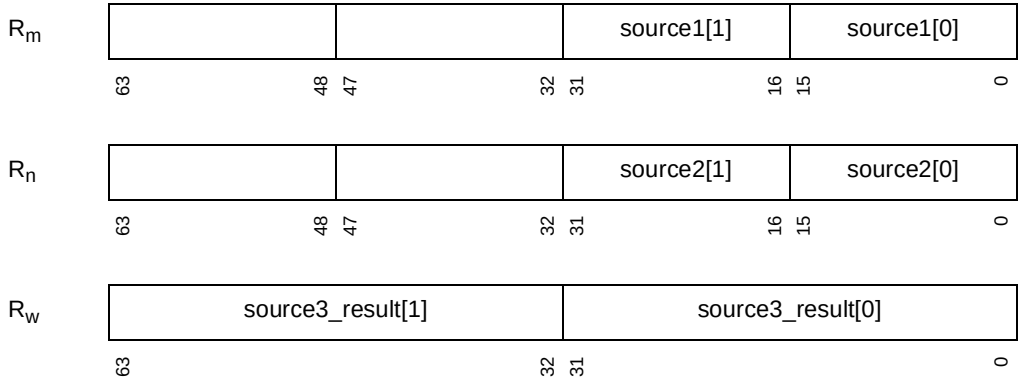



```
source1 ← MultiSignExtend16(Rm);
source2 ← MultiSignExtend16(Rn);
source3_result ← MultiSignExtend32(Rw);
REPEAT i FROM 0 FOR 2
{
   temp ← source1[i] × source2[i];
   temp ← SignedSaturate32(temp << 1);
   source3_result[i] ← SignedSaturate32(source3_result[i] - temp);
}
Rw ← MultiRegister32(source3_result);
```

Description:

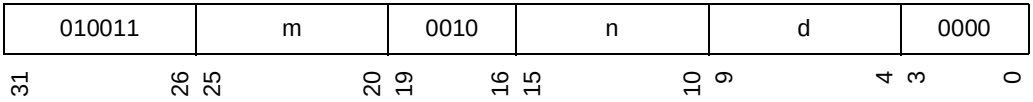
This multimedia instruction performs full-width, fractional multiplication on corresponding packed, signed 16-bit elements held in the lower halves of R_m and R_n, normalizes the results to a 32-bit fractional format, subtracts from corresponding packed, signed 32-bit elements held in R_w, saturates to the 32-bit signed fractional range, and places the packed 32-bit results in R_w. In the special case of a fractional multiply of -1 by -1, the multiplication result, which would otherwise be out of representable range, is saturated to the largest representable positive value before the subtraction and saturation stages. No rounding is necessary.

Multimedia Formats:



MMUL.L Rm, Rn, Rd

MMUL.L Rm, Rn, Rd

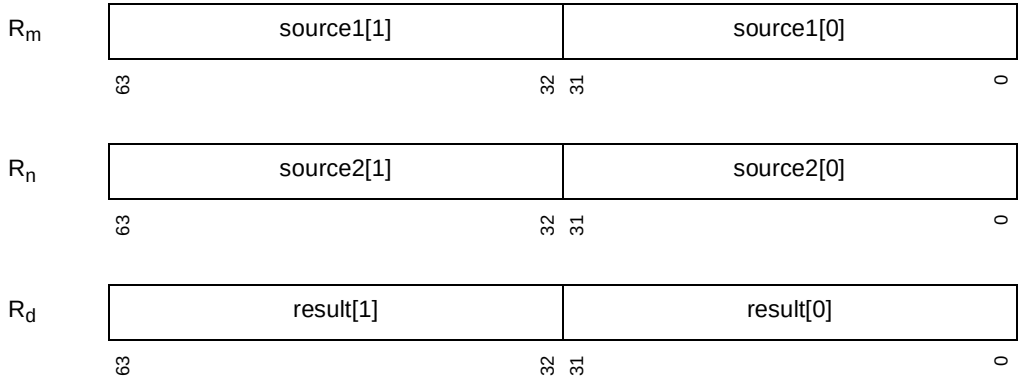



```
source1 ← MultiZeroExtend32(Rm);
source2 ← MultiZeroExtend32(Rn);
REPEAT i FROM 0 FOR 2
   result[i] ← ZeroExtend32(source1[i] × source2[i]);
Rd ← MultiRegister32(result);
```

Description:

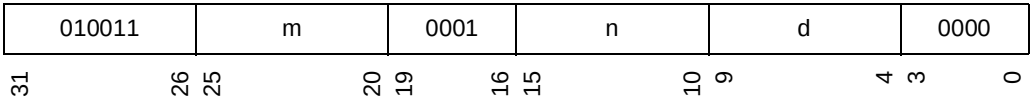
This multimedia instruction performs modulo, 32-bit multiplication on corresponding packed 32-bit elements held in R_m and R_n, and places the packed results in R_d. Sign is unimportant for modulo arithmetic and so this instruction can be used on both signed and unsigned types.

Multimedia Formats:



MMUL.W Rm, Rn, Rd

MMUL.W Rm, Rn, Rd

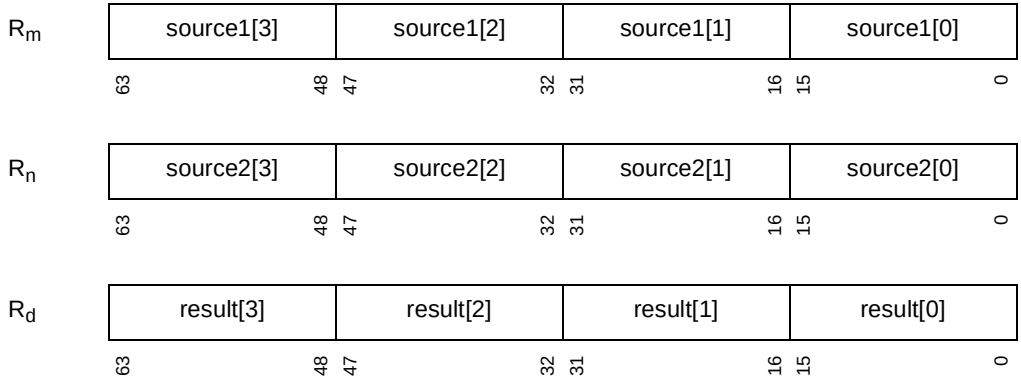



```
source1 ← MultiZeroExtend16(Rm);
source2 ← MultiZeroExtend16(Rn);
REPEAT i FROM 0 FOR 4
   result[i] ← ZeroExtend16(source1[i] × source2[i]);
Rd ← MultiRegister16(result);
```

Description:

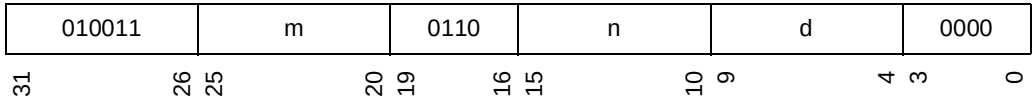
This multimedia instruction performs modulo, 16-bit multiplication on corresponding packed 16-bit elements held in R_m and R_n, and places the packed results in R_d. Sign is unimportant for modulo arithmetic and so this instruction can be used on both signed and unsigned types.

Multimedia Formats:



MMULFX.L Rm, Rn, Rd

MMULFX.L Rm, Rn, Rd

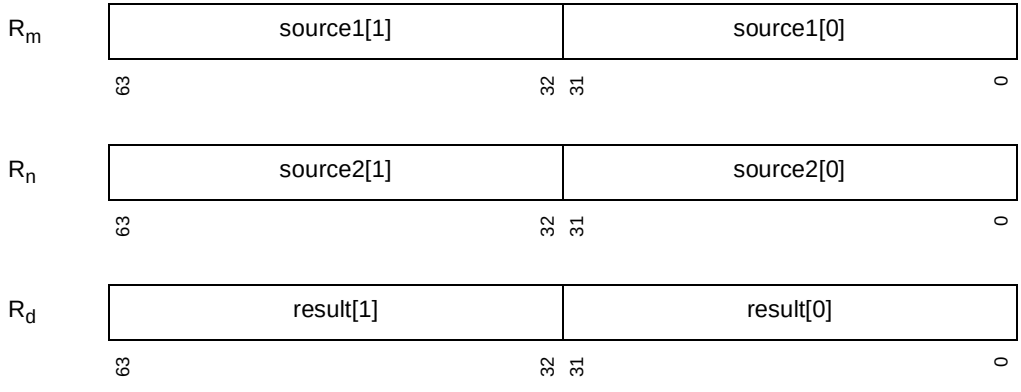



```
source1 ← MultiSignExtend32(Rm);
source2 ← MultiSignExtend32(Rn);
REPEAT i FROM 0 FOR 2
{
   temp ← source1[i] × source2[i];
   result[i] ← SignedSaturate32(temp >> 31);
}
Rd ← MultiRegister32(result);
```

Description:

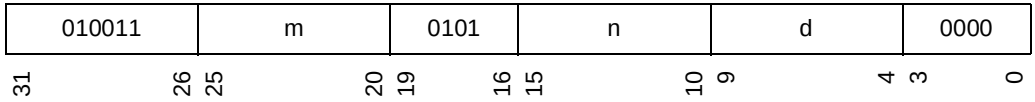
This multimedia instruction performs 32-bit fractional multiplication on corresponding packed, signed, 32-bit elements held in R_m and R_n, rounds the results back to the 32-bit fractional format, and places the packed results in R_d. The instruction provides rounding towards minus. In the special case of a fractional multiply of -1 by -1, the result, which would otherwise be out of representable range, is saturated to the largest representable positive value.

Multimedia Formats:



MMULFX.W Rm, Rn, Rd

MMULFX.W Rm, Rn, Rd

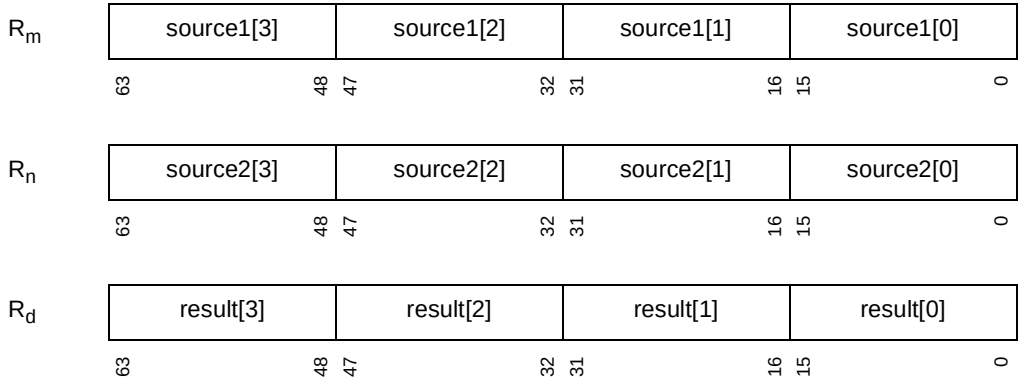



```
source1 ← MultiSignExtend16(Rm);
source2 ← MultiSignExtend16(Rn);
REPEAT i FROM 0 FOR 4
{
   temp ← source1[i] × source2[i];
   result[i] ← SignedSaturate16(temp >> 15);
}
Rd ← MultiRegister16(result);
```

Description:

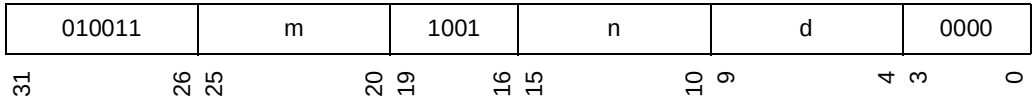
This multimedia instruction performs 16-bit fractional multiplication on corresponding packed, signed, 16-bit elements held in R_m and R_n, rounds the results back to the 16-bit fractional format, and places the packed results in R_d. The instruction provides rounding towards minus. In the special case of a fractional multiply of -1 by -1, the result, which would otherwise be out of representable range, is saturated to the largest representable positive value.

Multimedia Formats:



MMULFXRP.W Rm, Rn, Rd

MMULFXRP.W Rm, Rn, Rd

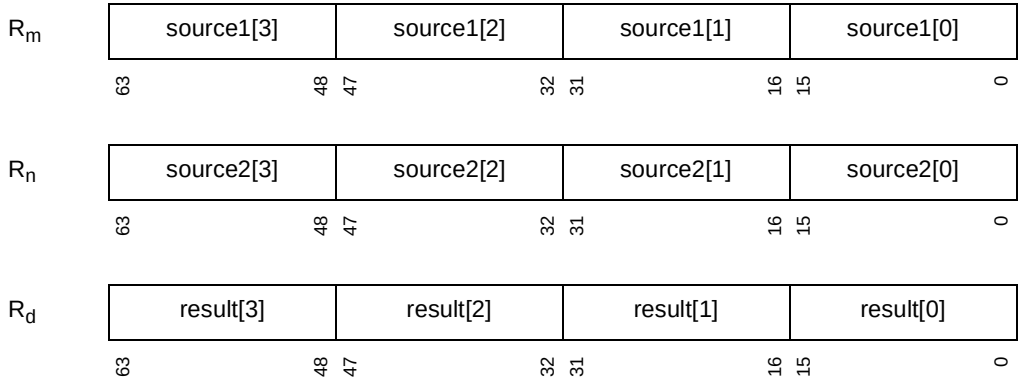



```
source1 ← MultiSignExtend16(Rm);
source2 ← MultiSignExtend16(Rn);
REPEAT i FROM 0 FOR 4
{
   temp ← source1[i] × source2[i];
   result[i] ← SignedSaturate16((temp + 2^14) >> 15);
}
Rd ← MultiRegister16(result);
```

Description:

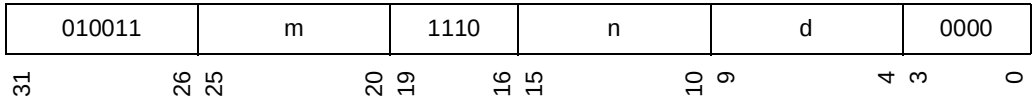
This multimedia instruction performs 16-bit fractional multiplication on corresponding packed, signed, 16-bit elements held in R_m and R_n, rounds the results back to the 16-bit fractional format, and places the packed results in R_d. The instruction provides rounding towards the nearest-positive. In the special case of a fractional multiply of -1 by -1, the result, which would otherwise be out of representable range, is saturated to the largest representable positive value.

Multimedia Formats:



MMULHI.WL Rm, Rn, Rd

MMULHI.WL Rm, Rn, Rd

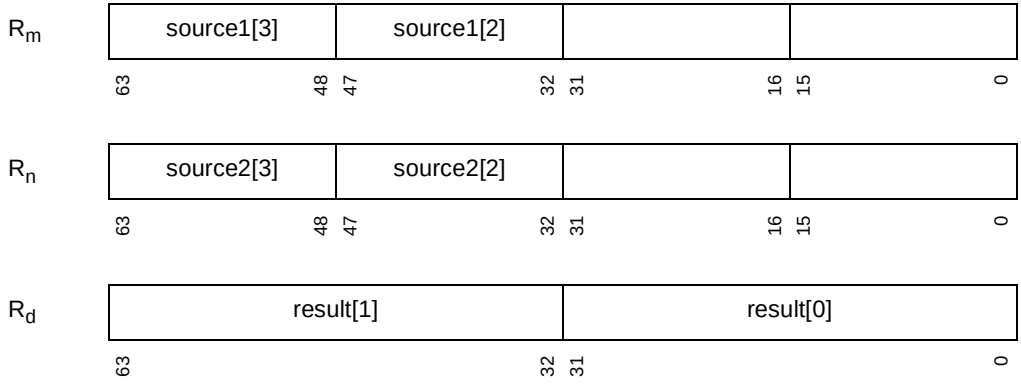



```
source1 ← MultiSignExtend16(Rm);
source2 ← MultiSignExtend16(Rn);
REPEAT i FROM 0 FOR 2
   result[i] ← source1[i + 2] × source2[i + 2];
Rd ← MultiRegister32(result);
```

Description:

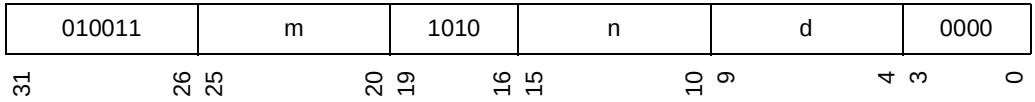
This multimedia instruction performs full-width multiplication on corresponding signed, packed, 16-bit elements held in the higher halves of R_m and R_n, and places the packed, 32-bit results in R_d. Element 0 of R_d contains the full-width multiplication of the two signed 16-bit values held in element 2 of R_m and in element 2 of R_n. Element 1 of R_d contains the full-width multiplication of the two signed 16-bit values held in element 3 of R_m and in element 3 of R_n.

Multimedia Formats:



MMULLO.WL Rm, Rn, Rd

MMULLO.WL Rm, Rn, Rd

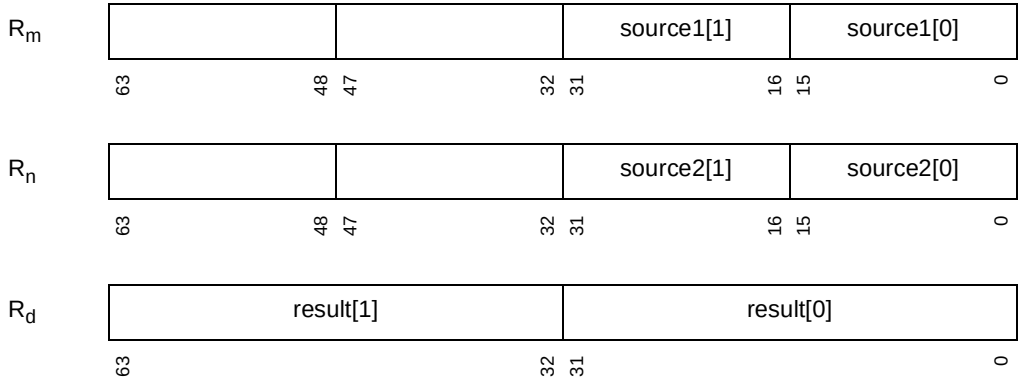



```
source1 ← MultiSignExtend16(Rm);
source2 ← MultiSignExtend16(Rn);
REPEAT i FROM 0 FOR 2
   result[i] ← source1[i] × source2[i];
Rd ← MultiRegister32(result);
```

Description:

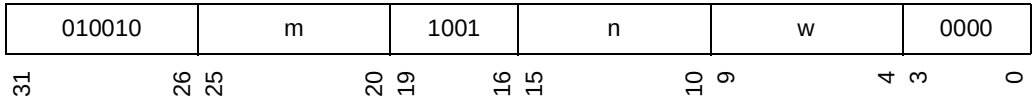
This multimedia instruction performs full-width multiplication on corresponding signed, packed, 16-bit elements held in the lower halves of R_m and R_n, and places the packed, 32-bit results in R_d. Element 0 of R_d contains the full-width multiplication of the two signed 16-bit values held in element 0 of R_m and in element 0 of R_n. Element 1 of R_d contains the full-width multiplication of the two signed 16-bit values held in element 1 of R_m and in element 1 of R_n.

Multimedia Formats:



MMULSUM.WQ Rm, Rn, Rw

MMULSUM.WQ Rm, Rn, Rw

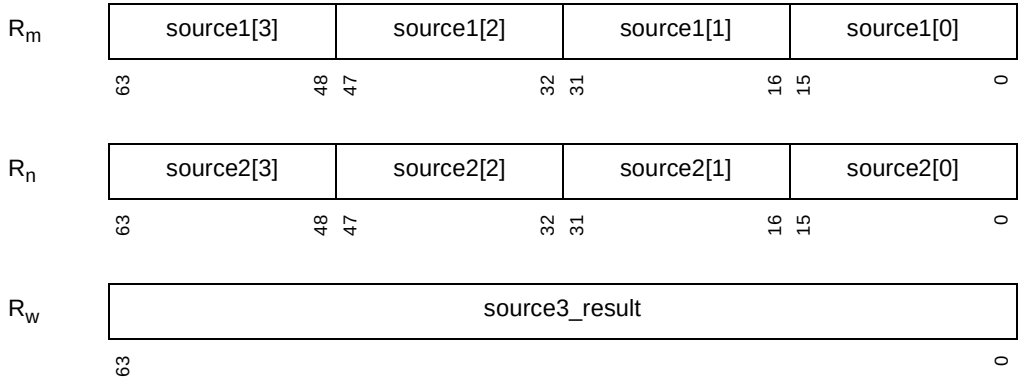



```
source1 ← MultiSignExtend16(Rm);
source2 ← MultiSignExtend16(Rn);
source3_result ← SignExtend64(Rw);
acc ← 0;
REPEAT i FROM 0 FOR 4
   acc ← acc + (source1[i] × source2[i]);
source3_result ← source3_result + acc;
Rw ← Register(source3_result);
```

Description:

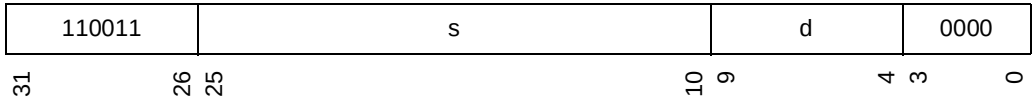
This multimedia instruction performs full-width multiplications on corresponding, packed, signed, 16-bit elements held in R_m and R_n, sums all 4 of these 32-bit intermediate results together, adds the total to the scalar 64-bit value held in R_w, and places the 64-bit result in R_w. The additions are performed using 64-bit modulo arithmetic.

Multimedia Formats:



MOVI imm, Rd

MOVI imm, Rd




```
imm ← SignExtend16(s);
result ← imm;
Rd ← Register(result);
```

Description:

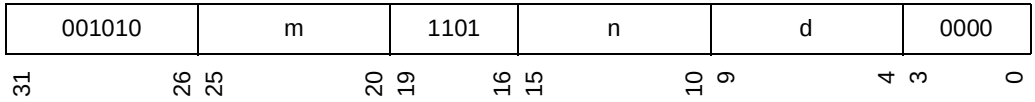
This instruction copies the sign-extended 16-bit immediate field s to R_d.

Notes:

The 'imm' in the assembly syntax represents the immediate s after sign extension.

MPERM.W Rm, Rn, Rd

MPERM.W Rm, Rn, Rd



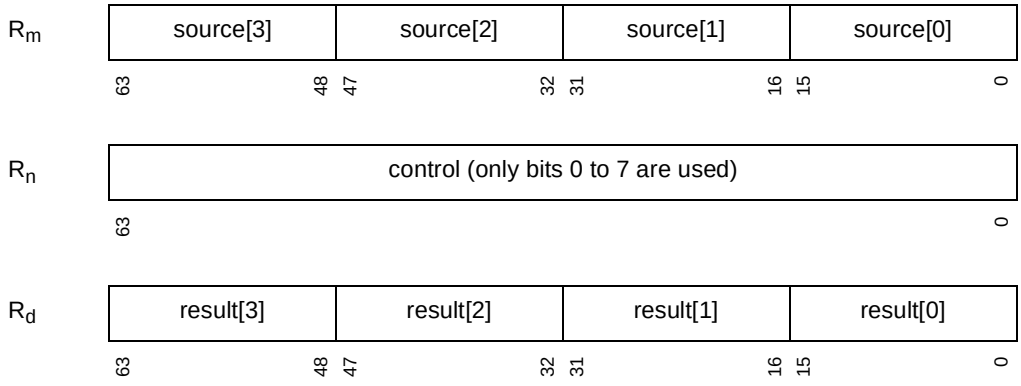

```
source ← MultiZeroExtend16(Rm);
control ← ZeroExtend8(Rn);
REPEAT i FROM 0 FOR 4
{
   index ← ZeroExtend2(control >> (i × 2));
   result[i] ← source[index];
}
Rd ← MultiRegister16(result);
```

Description:

This multimedia instruction permutes the packed 16-bit elements in R_m according to the control value held in the lowest 8 bits of R_n, and places the packed result in R_d. For each 16-bit element in the result, two bits from the control value determine which 16-bit element from the source is copied to that result element. The highest 56 bits of Rn are ignored.

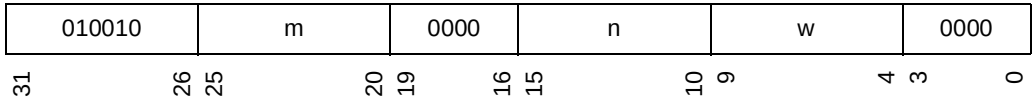
Bits of R_n	Interpretation
[0,1]	Selects which of the four 16-bit elements in R_m to place in element 0 of R_d
[2,3]	Selects which of the four 16-bit elements in R_m to place in element 1 of R_d
[4,5]	Selects which of the four 16-bit elements in R_m to place in element 2 of R_d
[6,7]	Selects which of the four 16-bit elements in R_m to place in element 3 of R_d
[8,63]	Ignored

Multimedia Formats:



MSAD.UBQ Rm, Rn, Rw

MSAD.UBQ Rm, Rn, Rw

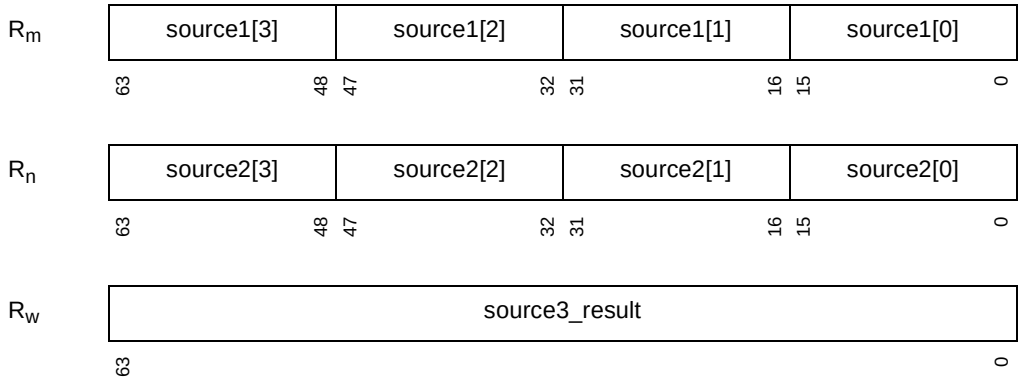



```
source1 ← MultiZeroExtend8(Rm);
source2 ← MultiZeroExtend8(Rn);
source3_result ← ZeroExtend64(Rw);
acc ← 0;
REPEAT i FROM 0 FOR 8
{
   temp ← source1[i] - source2[i];
   IF (temp < 0)
      temp ← - temp;
   acc ← acc + temp;
}
source3_result ← source3_result + acc;
Rw ← Register(source3_result);
```

Description:

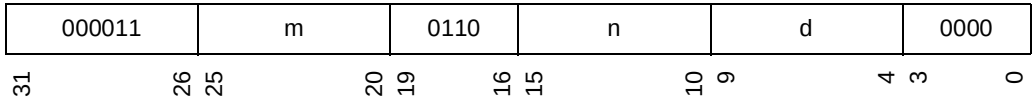
This multimedia instruction calculates the absolute-difference of corresponding packed, unsigned, 8-bit elements held in R_m and R_n, sums all 8 of these differences, adds the total to the scalar 64-bit value held in R_w, and places the 64-bit result in R_w. The additions are performed using 64-bit modulo arithmetic.

Multimedia Formats:



MSHALDS.L Rm, Rn, Rd

MSHALDS.L Rm, Rn, Rd

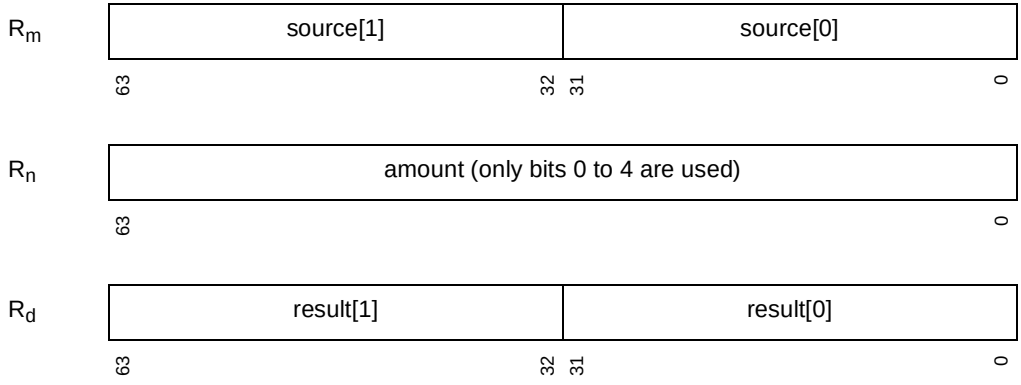



```
source ← MultiSignExtend32(Rm);
amount ← ZeroExtend64(Rn);
REPEAT i FROM 0 FOR 2
   result[i] ← SignedSaturate32(source[i] << ZeroExtend5(amount));
Rd ← MultiRegister32(result);
```

Description:

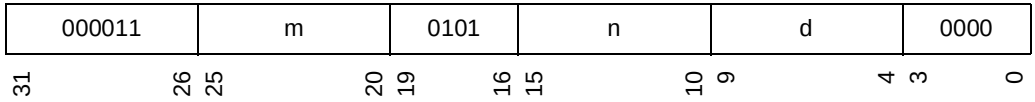
This multimedia instruction performs an arithmetic, saturating, left shift on each of the packed 32-bit elements held in R_m with a shift amount specified in the lowest 5 bits of R_n, and places the packed results in R_d. The highest 59 bits of R_n are ignored. The shifts are saturated to the signed range $[-2^{31}, 2^{31}]$.

Multimedia Formats:



MSHALDS.W Rm, Rn, Rd

MSHALDS.W Rm, Rn, Rd

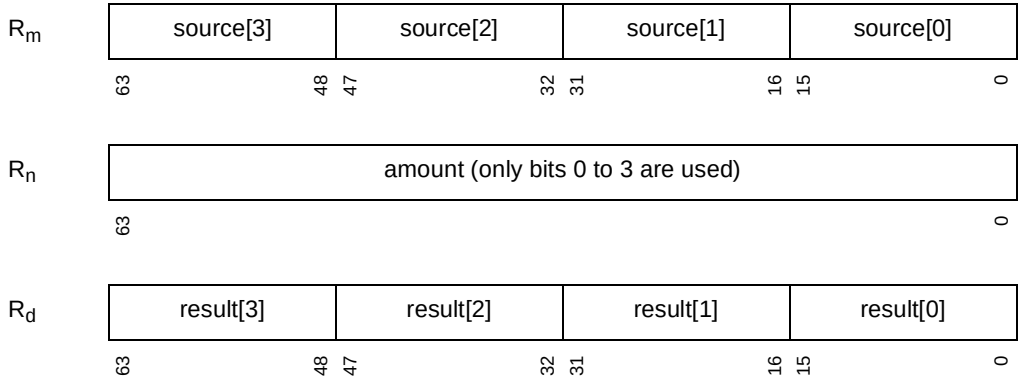



```
source ← MultiSignExtend16(Rm);
amount ← ZeroExtend64(Rn);
REPEAT i FROM 0 FOR 4
   result[i] ← SignedSaturate16(source[i] << ZeroExtend4(amount));
Rd ← MultiRegister16(result);
```

Description:

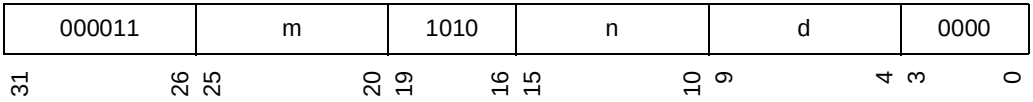
This multimedia instruction performs an arithmetic, saturating, left shift on each of the packed 16-bit elements held in R_m with a shift amount specified in the lowest 4 bits of R_n, and places the packed results in R_d. The highest 60 bits of R_n are ignored. The shifts are saturated to the signed range $[-2^{15}, 2^{15}]$.

Multimedia Formats:



MSHARD.L Rm, Rn, Rd

MSHARD.L Rm, Rn, Rd

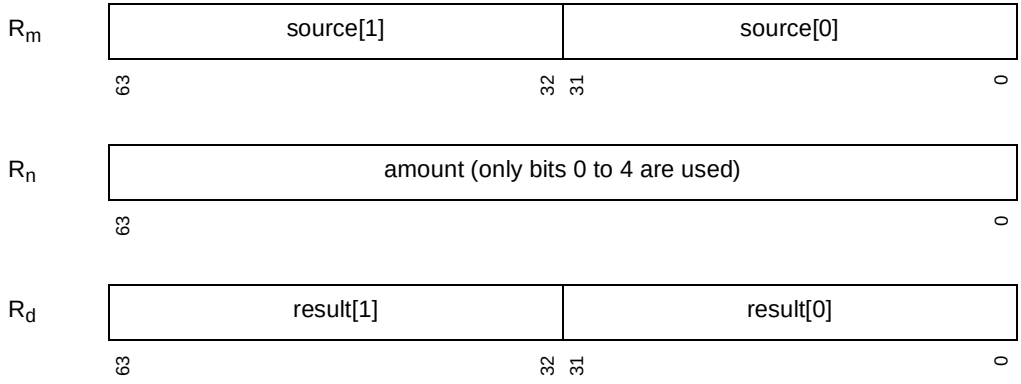



```
source ← MultiSignExtend32(Rm);
amount ← ZeroExtend64(Rn);
REPEAT i FROM 0 FOR 2
   result[i] ← SignExtend32(source[i] >> ZeroExtend5(amount));
Rd ← MultiRegister32(result);
```

Description:

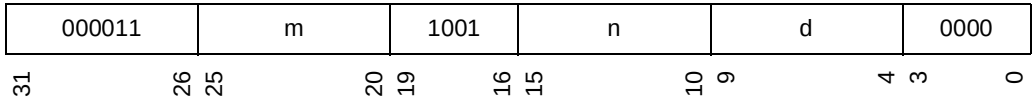
This multimedia instruction performs an arithmetic, right shift on each of the packed 32-bit elements held in R_m with a shift amount specified in the lowest 5 bits of R_n, and places the packed results in R_d. The highest 59 bits of R_n are ignored.

Multimedia Formats:



MSHARD.W Rm, Rn, Rd

MSHARD.W Rm, Rn, Rd

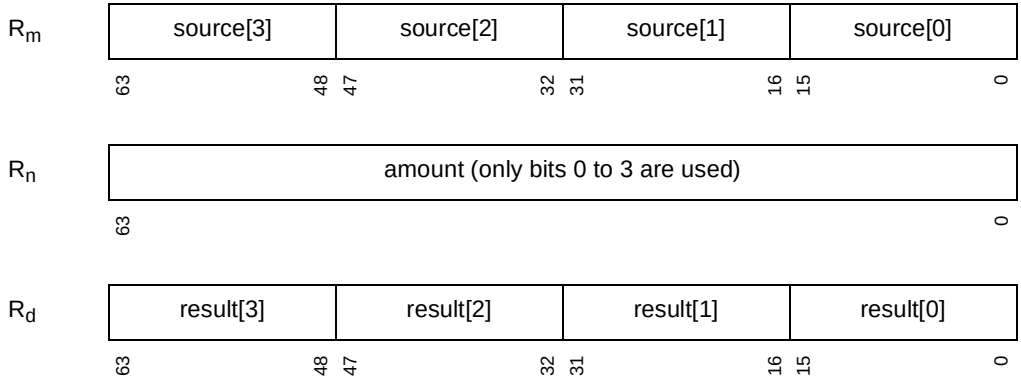



```
source ← MultiSignExtend16(Rm);
amount ← ZeroExtend64(Rn);
REPEAT i FROM 0 FOR 4
   result[i] ← SignExtend16(source[i] >> ZeroExtend4(amount));
Rd ← MultiRegister16(result);
```

Description:

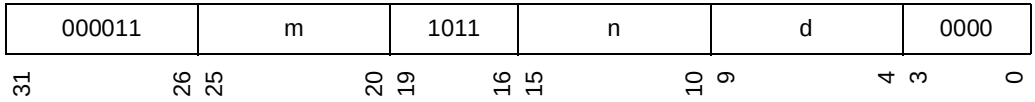
This multimedia instruction performs an arithmetic, right shift on each of the packed 16-bit elements held in R_m with a shift amount specified in the lowest 4 bits of R_n, and places the packed results in R_d. The highest 60 bits of R_n are ignored.

Multimedia Formats:



MSHARDS.Q Rm, Rn, Rd

MSHARDS.Q Rm, Rn, Rd

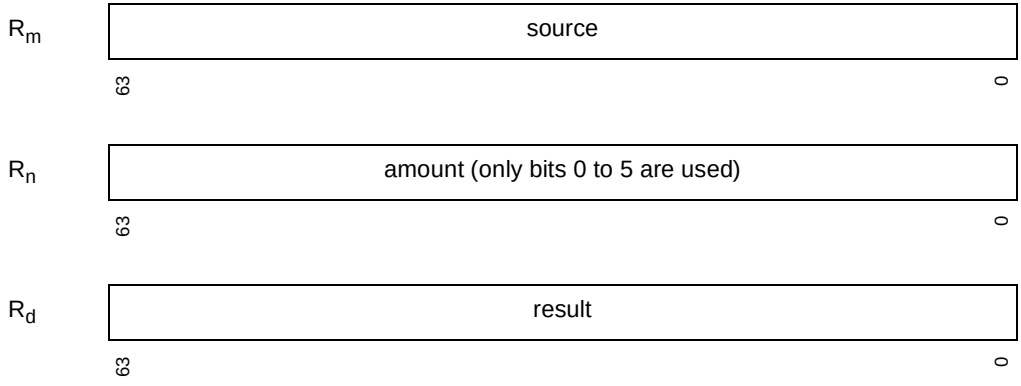



```
source ← SignExtend64(Rm);
amount ← ZeroExtend64(Rn);
result ← SignedSaturate16(source >> ZeroExtend6(amount));
Rd ← Register(result);
```

Description:

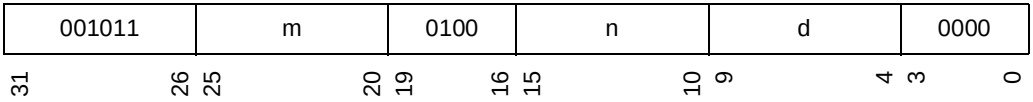
This multimedia instruction performs an arithmetic, right shift on the scalar value held in R_m with a shift amount specified in the lowest 6 bits of R_n, and places the result in R_d. The highest 58 bits of R_n are ignored. The shift is saturated to the signed range $[-2^{15}, 2^{15}]$.

Multimedia Formats:



MSHFHI.B Rm, Rn, Rd

MSHFHI.B Rm, Rn, Rd

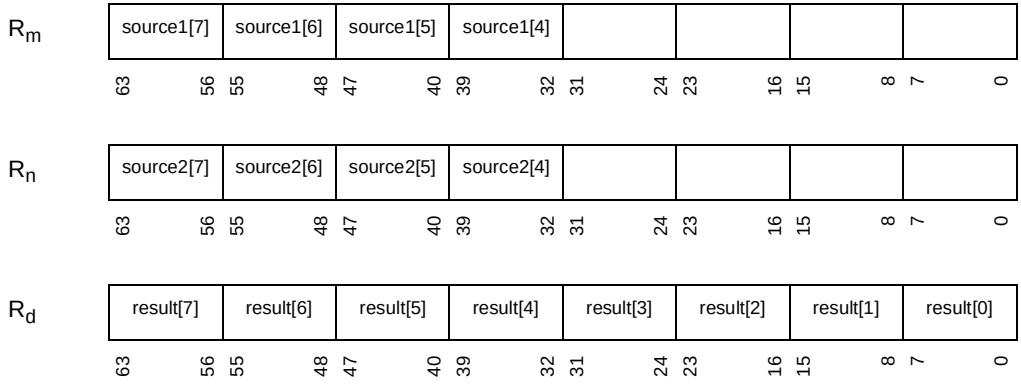



```
source1 ← MultiZeroExtend8(Rm);
source2 ← MultiZeroExtend8(Rn);
REPEAT i FROM 0 FOR 4
{
   result[i × 2] ← source1[i + 4];
   result[(i × 2) + 1] ← source2[i + 4];
}
Rd ← MultiRegister8(result);
```

Description:

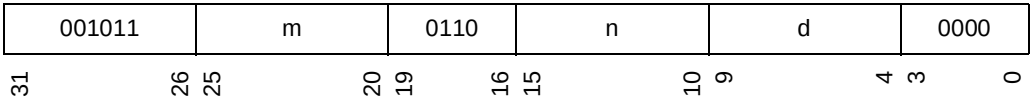
This multimedia instruction performs a shuffle on the packed 8-bit elements held in R_m and R_n, and produces higher-half results that are placed in R_d. The higher 4 elements of R_m are copied (in order) to even-numbered elements of the result, and the higher 4 elements of R_n are copied (in order) to odd-numbered elements of the result.

Multimedia Formats:



MSHFHI.L Rm, Rn, Rd

MSHFHI.L Rm, Rn, Rd

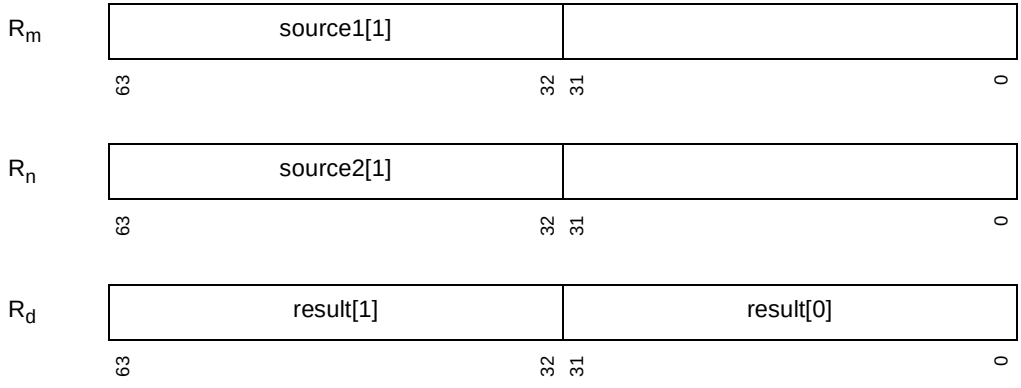


source1 ← MultiZeroExtend$_{32}$(R_m);
source2 ← MultiZeroExtend$_{32}$(R_n);
result[0] ← source1[1];
result[1] ← source2[1];
R_d ← MultiRegister$_{32}$(result);

Description:

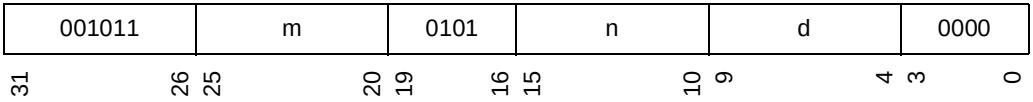
This multimedia instruction performs a shuffle on the packed 32-bit elements held in R_m and R_n, and produces higher-half results that are placed in R_d. The higher element of R_m is copied to the even-numbered element of the result, and the higher element of R_n is copied to the odd-numbered element of the result.

Multimedia Formats:



MSHFHI.W Rm, Rn, Rd

MSHFHI.W Rm, Rn, Rd

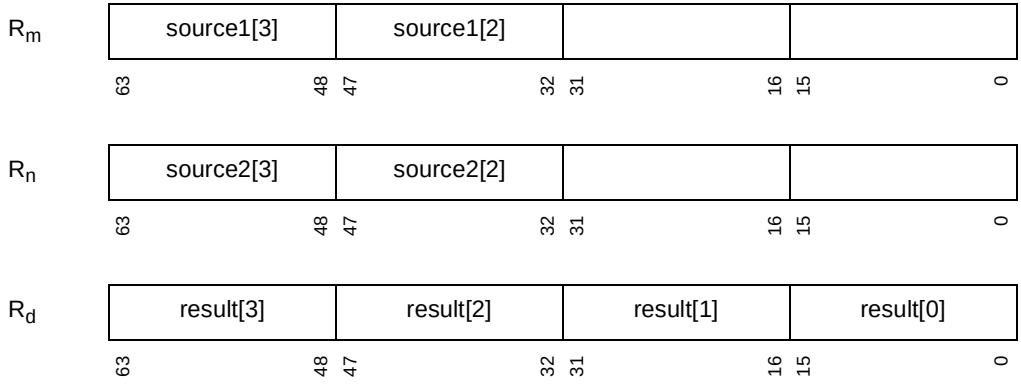



```
source1 ← MultiZeroExtend16(Rm);
source2 ← MultiZeroExtend16(Rn);
REPEAT i FROM 0 FOR 2
{
   result[i × 2] ← source1[i + 2];
   result[(i × 2) + 1] ← source2[i + 2];
}
Rd ← MultiRegister16(result);
```

Description:

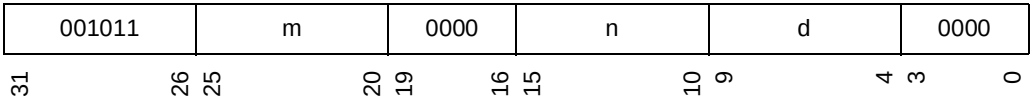
This multimedia instruction performs a shuffle on the packed 16-bit elements held in R_m and R_n, and produces higher-half results that are placed in R_d. The higher 2 elements of R_m are copied (in order) to even-numbered elements of the result, and the higher 2 elements of R_n are copied (in order) to odd-numbered elements of the result.

Multimedia Formats:



MSHFLO.B Rm, Rn, Rd

MSHFLO.B Rm, Rn, Rd

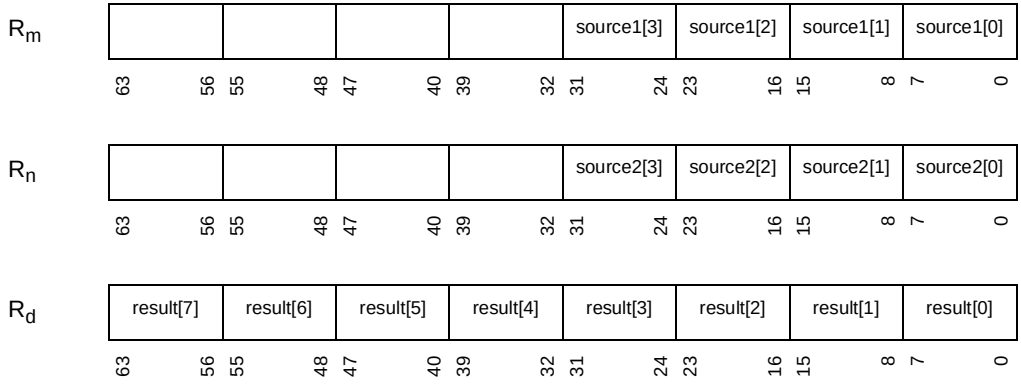



```
source1 ← MultiZeroExtend8(Rm);
source2 ← MultiZeroExtend8(Rn);
REPEAT i FROM 0 FOR 4
{
   result[i × 2] ← source1[i];
   result[(i × 2) + 1] ← source2[i];
}
Rd ← MultiRegister8(result);
```

Description:

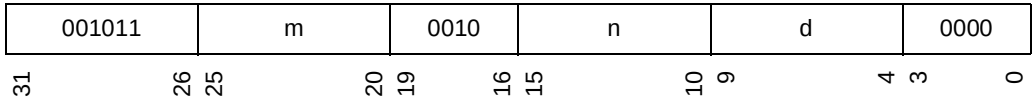
This multimedia instruction performs a shuffle on the packed 8-bit elements held in R_m and R_n, and produces lower-half results that are placed in R_d. The lower 4 elements of R_m are copied (in order) to even-numbered elements of the result, and the lower 4 elements of R_n are copied (in order) to odd-numbered elements of the result.

Multimedia Formats:



MSHFLO.L Rm, Rn, Rd

MSHFLO.L Rm, Rn, Rd

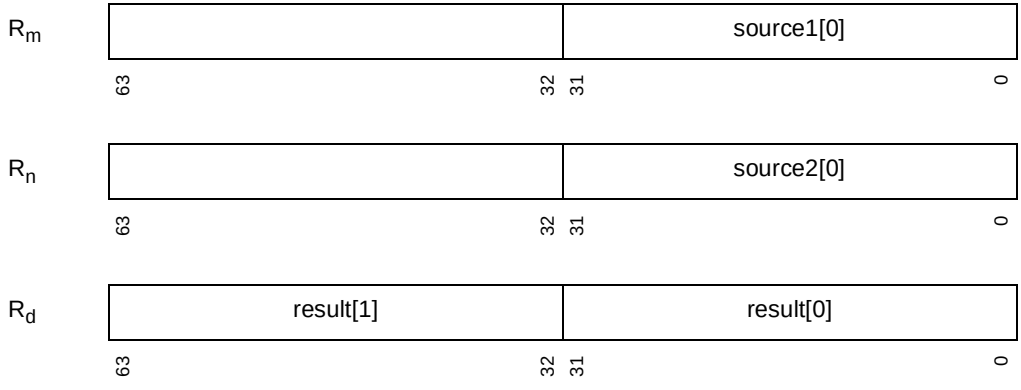



```
source1 ← MultiZeroExtend32(Rm);
source2 ← MultiZeroExtend32(Rn);
result[0] ← source1[0];
result[1] ← source2[0];
Rd ← MultiRegister32(result);
```

Description:

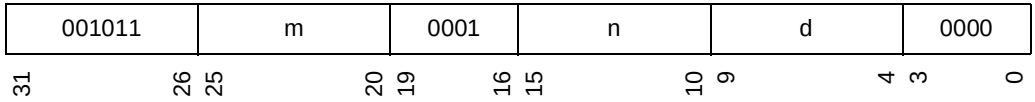
This multimedia instruction performs a shuffle on the packed 32-bit elements held in R_m and R_n, and produces lower-half results that are placed in R_d. The lower element of R_m is copied to the even-numbered element of the result, and the lower element of R_n is copied to the odd-numbered element of the result.

Multimedia Formats:



MSHFLO.W Rm, Rn, Rd

MSHFLO.W Rm, Rn, Rd

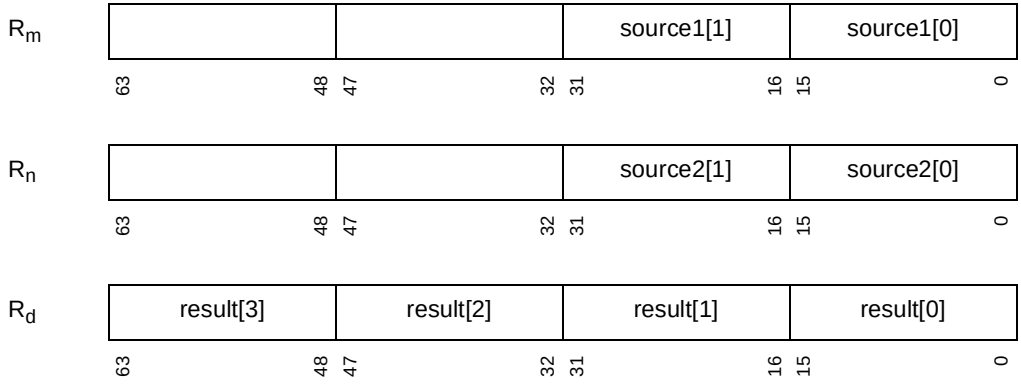



```
source1 ← MultiZeroExtend16(Rm);
source2 ← MultiZeroExtend16(Rn);
REPEAT i FROM 0 FOR 2
{
   result[i × 2] ← source1[i];
   result[(i × 2) + 1] ← source2[i];
}
Rd ← MultiRegister16(result);
```

Description:

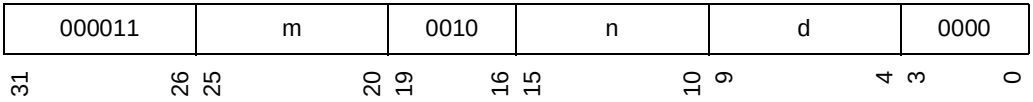
This multimedia instruction performs a shuffle on the packed 16-bit elements held in R_m and R_n, and produces lower-half results that are placed in R_d. The lower 2 elements of R_m are copied (in order) to even-numbered elements of the result, and the lower 2 elements of R_n are copied (in order) to odd-numbered elements of the result.

Multimedia Formats:



MSHLLD.L Rm, Rn, Rd

MSHLLD.L Rm, Rn, Rd

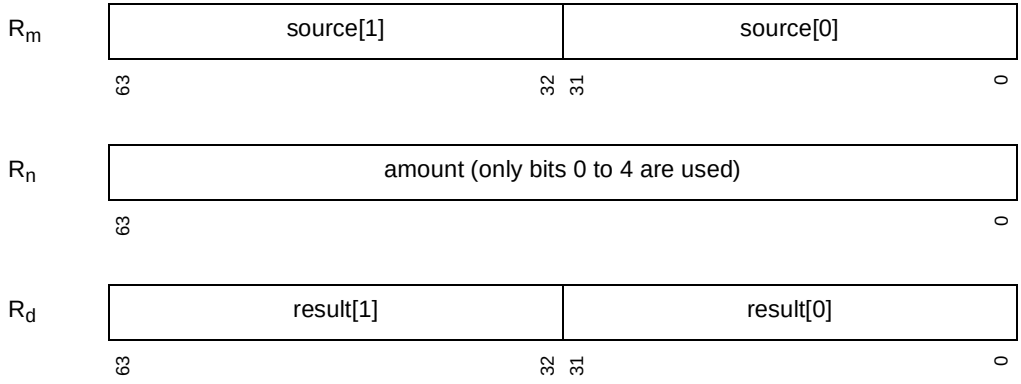



```
source ← MultiZeroExtend32(Rm);
amount ← ZeroExtend64(Rn);
REPEAT i FROM 0 FOR 2
   result[i] ← ZeroExtend32(source[i] << ZeroExtend5(amount));
Rd ← MultiRegister32(result);
```

Description:

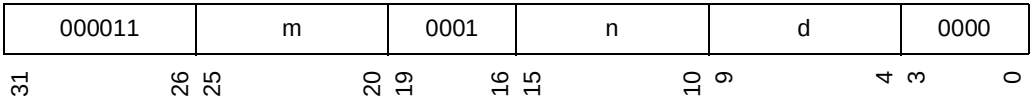
This multimedia instruction performs a logical left shift on each of the packed 32-bit elements held in R_m with a shift amount specified in the lowest 5 bits of R_n, and places the packed results in R_d. The highest 59 bits of R_n are ignored.

Multimedia Formats:



MSHLLD.W Rm, Rn, Rd

MSHLLD.W Rm, Rn, Rd

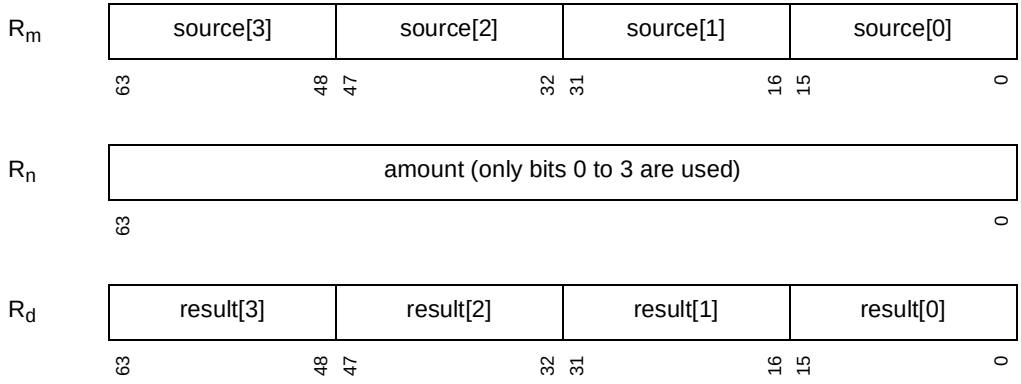



```
source ← MultiZeroExtend16(Rm);
amount ← ZeroExtend64(Rn);
REPEAT i FROM 0 FOR 4
   result[i] ← ZeroExtend16(source[i] << ZeroExtend4(amount));
Rd ← MultiRegister16(result);
```

Description:

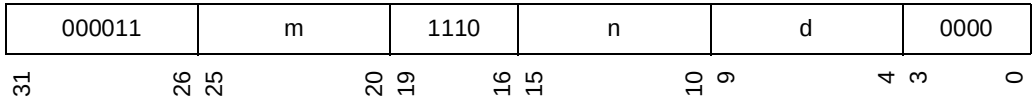
This multimedia instruction performs a logical left shift on each of the packed 16-bit elements held in R_m with a shift amount specified in the lowest 4 bits of R_n, and places the packed results in R_d. The highest 60 bits of R_n are ignored.

Multimedia Formats:



MSHLRD.L Rm, Rn, Rd

MSHLRD.L Rm, Rn, Rd

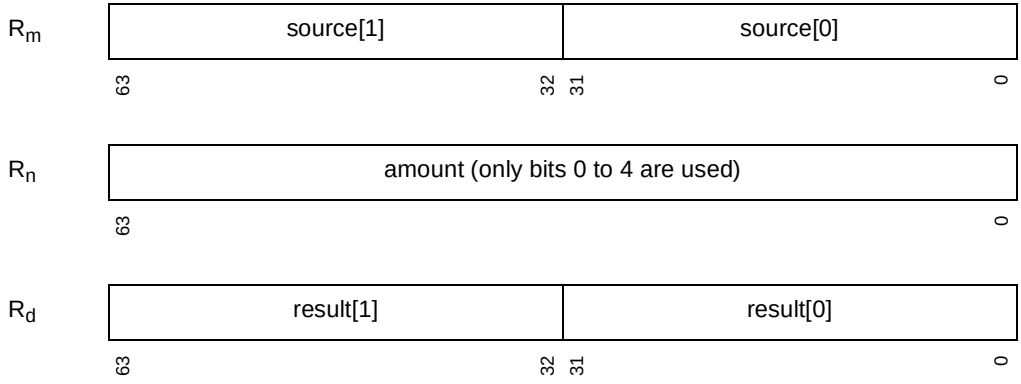



```
source ← MultiZeroExtend32(Rm);
amount ← ZeroExtend64(Rn);
REPEAT i FROM 0 FOR 2
   result[i] ← ZeroExtend32(source[i] >> ZeroExtend5(amount));
Rd ← MultiRegister32(result);
```

Description:

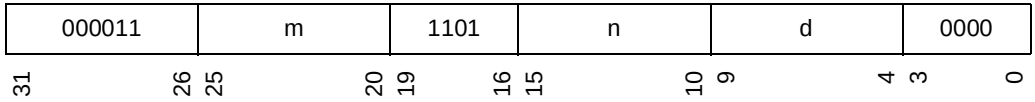
This multimedia instruction performs a logical right shift on each of the packed 32-bit elements held in R_m with a shift amount specified in the lowest 5 bits of R_n, and places the packed results in R_d. The highest 59 bits of R_n are ignored.

Multimedia Formats:



MSHLRD.W Rm, Rn, Rd

MSHLRD.W Rm, Rn, Rd

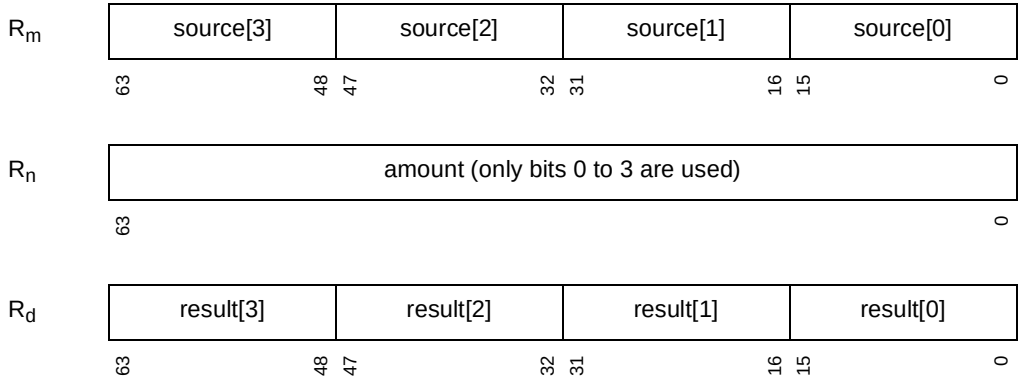



```
source ← MultiZeroExtend16(Rm);
amount ← ZeroExtend64(Rn);
REPEAT i FROM 0 FOR 4
   result[i] ← ZeroExtend16(source[i] >> ZeroExtend4(amount));
Rd ← MultiRegister16(result);
```

Description:

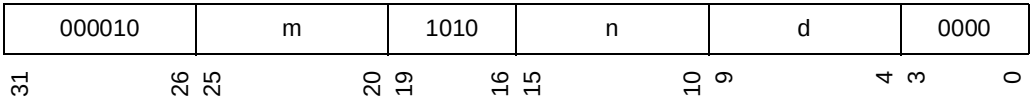
This multimedia instruction performs a logical right shift on each of the packed 16-bit elements held in R_m with a shift amount specified in the lowest 4 bits of R_n, and places the packed results in R_d. The highest 60 bits of R_n are ignored.

Multimedia Formats:



MSUB.L Rm, Rn, Rd

MSUB.L Rm, Rn, Rd

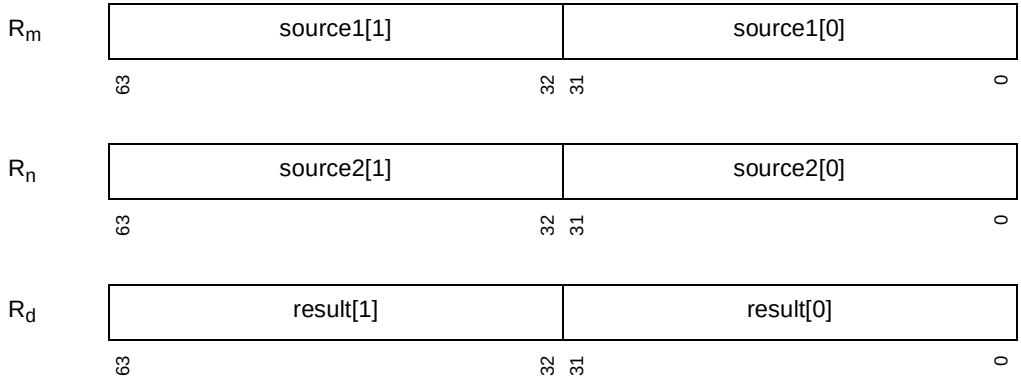



```
source1 ← MultiZeroExtend32(Rm);
source2 ← MultiZeroExtend32(Rn);
REPEAT i FROM 0 FOR 2
   result[i] ← ZeroExtend32(source1[i] - source2[i]);
Rd ← MultiRegister32(result);
```

Description:

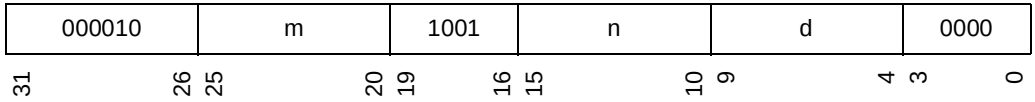
This multimedia instruction performs modulo, 32-bit subtraction on corresponding packed 32-bit elements held in R_m and R_n, and places the packed results in R_d. Sign is unimportant for modulo arithmetic and so this instruction can be used on both signed and unsigned types.

Multimedia Formats:



MSUB.W Rm, Rn, Rd

MSUB.W Rm, Rn, Rd

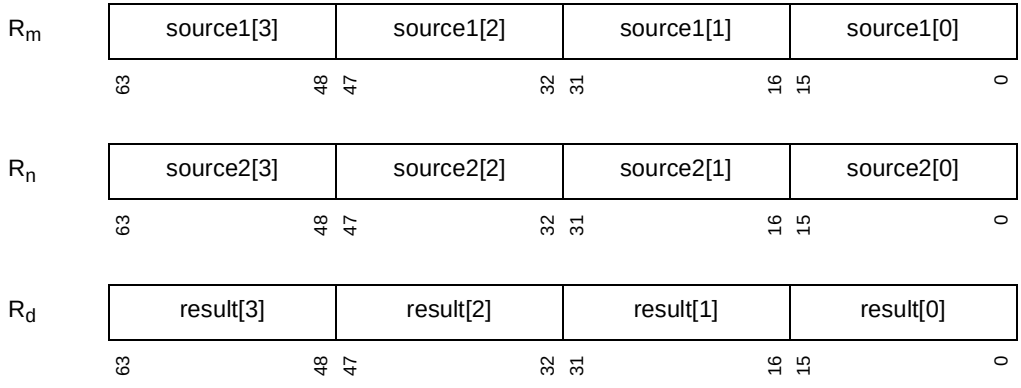



```
source1 ← MultiZeroExtend16(Rm);
source2 ← MultiZeroExtend16(Rn);
REPEAT i FROM 0 FOR 4
   result[i] ← ZeroExtend16(source1[i] - source2[i]);
Rd ← MultiRegister16(result);
```

Description:

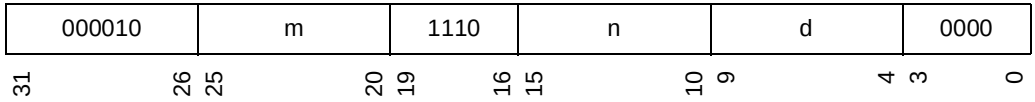
This multimedia instruction performs modulo, 16-bit subtraction on corresponding packed 16-bit elements held in R_m and R_n, and places the packed results in R_d. Sign is unimportant for modulo arithmetic and so this instruction can be used on both signed and unsigned types.

Multimedia Formats:



MSUBS.L Rm, Rn, Rd

MSUBS.L Rm, Rn, Rd

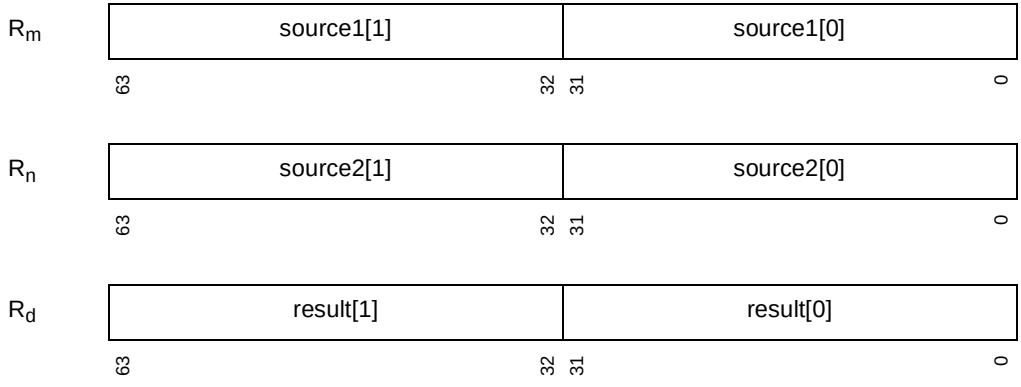



```
source1 ← MultiSignExtend32(Rm);
source2 ← MultiSignExtend32(Rn);
REPEAT i FROM 0 FOR 2
   result[i] ← SignedSaturate32(source1[i] - source2[i]);
Rd ← MultiRegister32(result);
```

Description:

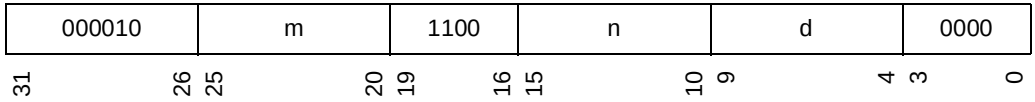
This multimedia instruction performs saturating, signed, 32-bit subtraction on corresponding packed 32-bit elements held in R_m and R_n, and places the packed results in R_d. The additions are saturated to the signed range $[-2^{31}, 2^{31}]$.

Multimedia Formats:



MSUBS.UB Rm, Rn, Rd

MSUBS.UB Rm, Rn, Rd

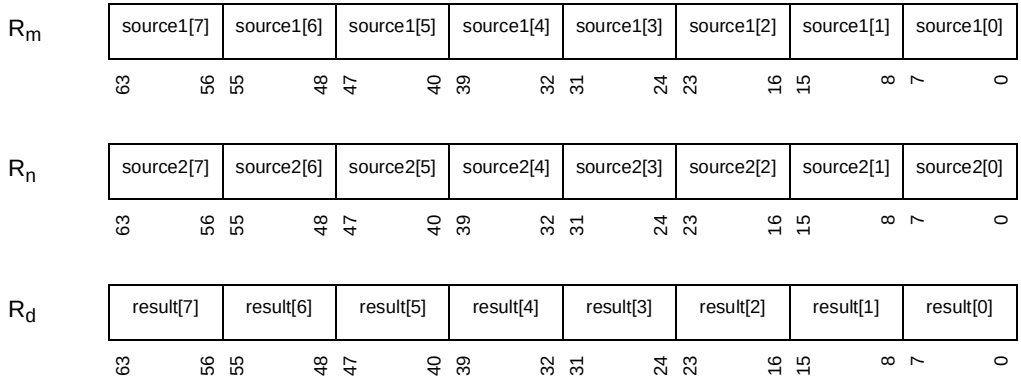



```
source1 ← MultiZeroExtend8(Rm);
source2 ← MultiZeroExtend8(Rn);
REPEAT i FROM 0 FOR 8
   result[i] ← UnsignedSaturate8(source1[i] - source2[i]);
Rd ← MultiRegister8(result);
```

Description:

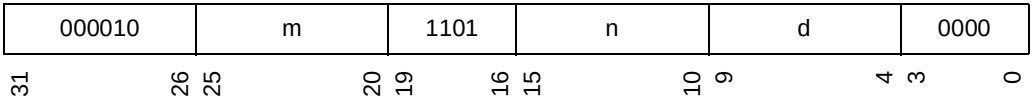
This multimedia instruction performs saturating, unsigned, 8-bit subtraction on corresponding packed 8-bit elements held in R_m and R_n, and places the packed results in R_d. The additions are saturated to the unsigned range [0, 256].

Multimedia Formats:



MSUBS.W Rm, Rn, Rd

MSUBS.W Rm, Rn, Rd

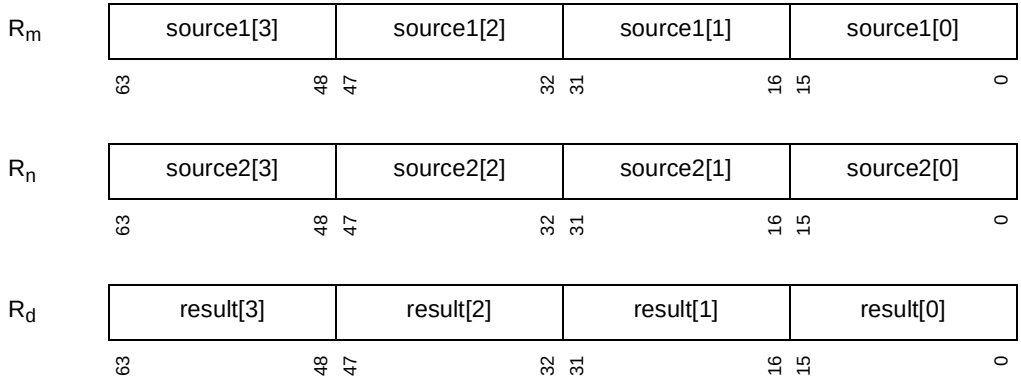



```
source1 ← MultiSignExtend16(Rm);
source2 ← MultiSignExtend16(Rn);
REPEAT i FROM 0 FOR 4
   result[i] ← SignedSaturate16(source1[i] - source2[i]);
Rd ← MultiRegister16(result);
```

Description:

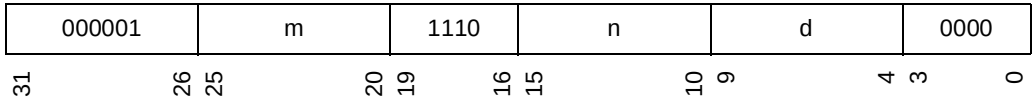
This multimedia instruction performs saturating, signed, 16-bit subtraction on corresponding packed 16-bit elements held in R_m and R_n, and places the packed results in R_d. The additions are saturated to the signed range $[-2^{15}, 2^{15}]$.

Multimedia Formats:



MULS.L Rm, Rn, Rd

MULS.L Rm, Rn, Rd



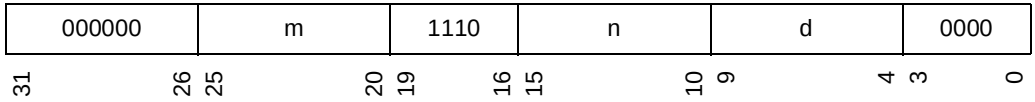
source1 ← SignExtend$_{32}$(R_m);
source2 ← SignExtend$_{32}$(R_n);
result ← source1 × source2;
R_d ← Register(result);

Description:

This instruction multiplies the signed lowest 32 bits of R_m by the signed lowest 32 bits of R_n and places the full 64-bit value of the result in R_d. The highest 32 bits of R_m and the highest 32 bits of R_n are ignored.

MULU.L Rm, Rn, Rd

MULU.L Rm, Rn, Rd



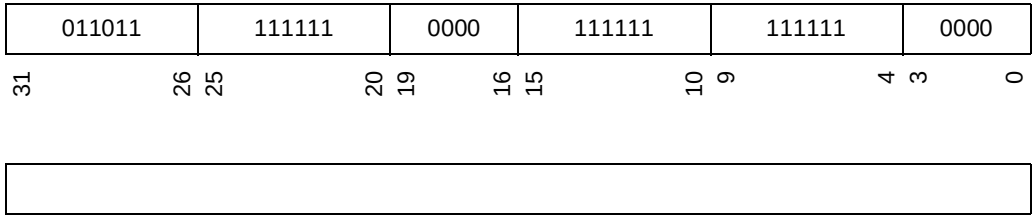
source1 ← ZeroExtend$_{32}$(R_m);
source2 ← ZeroExtend$_{32}$(R_n);
result ← source1 × source2;
R_d ← Register(result);

Description:

This instruction multiplies the unsigned lowest 32 bits of R_m by the unsigned lowest 32 bits of R_n and places the full 64-bit value of the result in R_d. The highest 32 bits of R_m and the highest 32 bits of R_n are ignored.

NOP

NOP

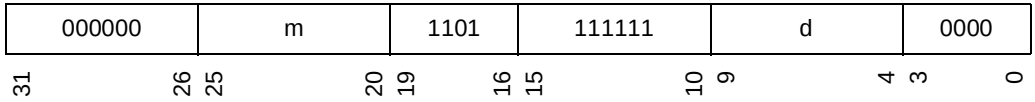


Description:

This instruction performs no operation.

NSB Rm, Rd

NSB Rm, Rd



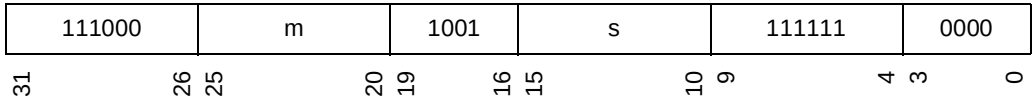

```
source ← SignExtend64(Rm);
REPEAT i FROM 0 FOR 64
{
   n ← 64 - i;
   IF (SignExtendn(source) = source)
      result ← i;
}
Rd ← Register(result);
```

Description:

This instruction counts the number of consecutive sign bits in R_m, subtracts one and places the result in R_d.

OCBI Rm, disp

OCBI Rm, disp




```
base ← ZeroExtend64(Rm);
disp ← SignExtend6(s) << 5;
address ← ZeroExtend64(base + disp);
IF (MalformedAddress(address))
   THROW WADDERR, address;
IF (MMU() AND DataAccessMiss(address))
   THROW WTLBMISS, address;
IF (MMU() AND WriteProhibited(address))
   THROW WRITEPROT, address;
OCBI(address);
```

Description:

This instruction invalidates an operand cache block (if any) that corresponds to a specified effective address. If the data in the operand cache block is dirty, it is discarded without write-back to memory.

The effective address is calculated by adding R_m to the sign-extended 6-bit immediate s multiplied by 32. The scaling factor is fixed at 32 regardless of the cache block size. There is no misalignment check on this instruction, and the calculated effective address can be any byte address. The calculated effective address is automatically aligned downwards to the nearest exact multiple of the cache block size. The effective address identifies a surrounding block of memory, which starts at an address aligned to the cache block size and has a size equal to the cache block size. The cache block size is implementation dependent.

OCBI checks for address error, translation miss and protection exception cases.

OCBI invalidates an implementation-dependent amount of data. For compatibility with other implementations, software must exercise care when using OCBI.

Explicit synchronization instructions are required to synchronize the effects of cache coherency instructions. SYNCO must be used to guarantee that previous OCBI instructions have completed their operation on the operand cache.

After completion, assuming no exception was raised, it is guaranteed that the targeted memory block in physical address space is not present in any operand or unified cache.

The behavior of this instruction when the MMU is disabled is described in *Volume 1, Chapter 6: SHmedia memory instructions*.

Possible exceptions:

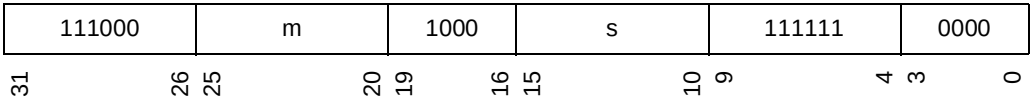
WADDERR, WTLBMISS, WRITEPROT

Notes:

The ‘disp’ in the assembly syntax represents the immediate s after sign extension and scaling.

OCBP Rm, disp

OCBP Rm, disp




```
base ← ZeroExtend64(Rm);
disp ← SignExtend6(s) << 5;
address ← ZeroExtend64(base + disp);
IF (MalformedAddress(address))
   THROW RADDERR, address;
IF (MMU() AND DataAccessMiss(address))
   THROW RTLBMISS, address;
IF (MMU() AND (ReadProhibited(address) AND WriteProhibited(address)))
   THROW READPROT, address;
OCBP(address);
```

Description:

This instruction purges an operand cache block (if any) that corresponds to a specified effective address. If the data in the operand cache block is dirty, it is written back to memory before being discarded.

The effective address is calculated by adding R_m to the sign-extended 6-bit immediate s multiplied by 32. The scaling factor is fixed at 32 regardless of the cache block size. There is no misalignment check on this instruction, and the calculated effective address can be any byte address. The calculated effective address is automatically aligned downwards to the nearest exact multiple of the cache block size. The effective address identifies a surrounding block of memory, which starts at an address aligned to the cache block size and has a size equal to the cache block size. The cache block size is implementation dependent.

OCBP checks for address error, translation miss and protection exception cases.

Explicit synchronization instructions are required to synchronize the effects of cache coherency instructions. SYNCO must be used to guarantee that previous OCBP instructions have completed their operation on the operand cache and on memory.

After completion, assuming no exception was raised, it is guaranteed that the targeted memory block in physical address space is not present in any operand or unified cache.

The behavior of this instruction when the MMU is disabled is described in *Volume 1, Chapter 6: SHmedia memory instructions*.

Possible exceptions:

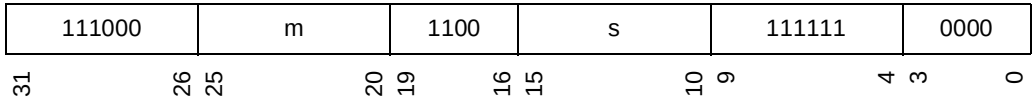
RADDERR, RTLBMISS, READPROT

Notes:

The ‘disp’ in the assembly syntax represents the immediate s after sign extension and scaling.

OCBWB Rm, disp

OCBWB Rm, disp




```
base ← ZeroExtend64(Rm);
disp ← SignExtend6(s) << 5;
address ← ZeroExtend64(base + disp);
IF (MalformedAddress(address))
   THROW RADDERR, address;
IF (MMU() AND DataAccessMiss(address))
   THROW RTLBMISS, address;
IF (MMU() AND (ReadProhibited(address) AND WriteProhibited(address)))
   THROW READPROT, address;
OCBWB(address);
```

Description:

This instruction write-backs an operand cache block (if any) that corresponds to a specified effective address. If the data in the operand cache block is dirty, it is written back to memory but is not discarded.

The effective address is calculated by adding R_m to the sign-extended 6-bit immediate s multiplied by 32. The scaling factor is fixed at 32 regardless of the cache block size. There is no misalignment check on this instruction, and the calculated effective address can be any byte address. The calculated effective address is automatically aligned downwards to the nearest exact multiple of the cache block size. The effective address identifies a surrounding block of memory, which starts at an address aligned to the cache block size and has a size equal to the cache block size. The cache block size is implementation dependent.

OCBWB checks for address error, translation miss and protection exception cases.

Explicit synchronization instructions are required to synchronize the effects of cache coherency instructions. SYNCO must be used to guarantee that previous OCBWB instructions have completed their operation on the operand cache and on memory.

After completion, assuming no exception was raised, it is guaranteed that the targeted memory block in physical address space will not be dirty in any operand or unified cache.

The behavior of this instruction when the MMU is disabled is described in *Volume 1, Chapter 6: SHmedia memory instructions*.

Possible exceptions:

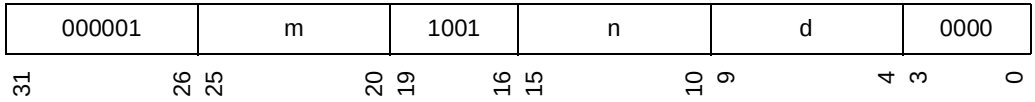
RADDERR, RTLBMISS, READPROT

Notes:

The ‘disp’ in the assembly syntax represents the immediate s after sign extension and scaling.

OR Rm, Rn, Rd

OR Rm, Rn, Rd



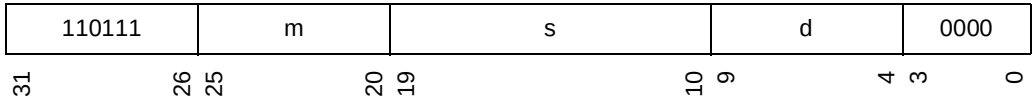

```
source1 ← SignExtend64(Rm);
source2 ← SignExtend64(Rn);
result ← source1 ∨ source2;
Rd ← Register(result);
```

Description:

This instruction performs a bitwise OR of R_m with R_n and places the result in R_d.

ORI Rm, imm, Rd

ORI Rm, imm, Rd



source1 ← SignExtend$_{64}$(R_m);
imm ← SignExtend$_{10}$(s);
result ← source1 ∨ imm;
R_d ← Register(result);

Description:

This instruction performs a bitwise OR of R_m with the sign-extended 10-bit immediate s and places the result in R_d.

There is no dedicated instruction for moving one general-purpose register value into another. It is recommended that the ORI instruction is used with an immediate value of 0:

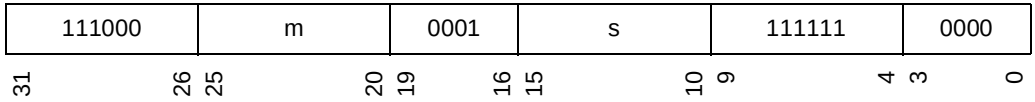
```
ORI Rm, 0, Rd ; move Rm into Rd
```

Notes:

The 'imm' in the assembly syntax represents the immediate s after sign extension.

PREFI Rm, disp

PREFI Rm, disp




```
base ← ZeroExtend64(Rm);
disp ← SignExtend6(s) << 5;
address ← ZeroExtend64(base + disp);
IF (NOT MalformedAddress(address))
   IF (NOT (MMU() AND InstPrefetchMiss(address)))
      IF (NOT (MMU() AND ExecuteProhibited(address)))
         PREFI(address);
```

Description:

This instruction indicates a software-directed instruction prefetch from a specified effective address. Software can use this instruction to give advance notice that particular instructions will be required. It is implementation-specific as to whether a prefetch will be performed.

The effective address is calculated by adding R_m to the sign-extended 6-bit immediate s multiplied by 32. The scaling factor is fixed at 32 regardless of the cache block size. There is no misalignment check on this instruction, and the calculated effective address can be any byte address. The calculated effective address is automatically aligned downwards to the nearest exact multiple of the cache block size. The effective address identifies a surrounding block of memory, which starts at an address aligned to the cache block size and has a size equal to the cache block size. The cache block size is implementation dependent.

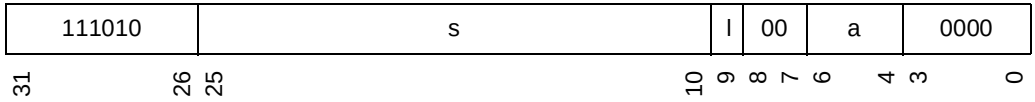
In exceptional cases, no exception is raised and the prefetch has no effect.

Notes:

The ‘disp’ in the assembly syntax represents the immediate s after sign extension and scaling.

PTA label, TRa

PTA label, TRa




```
pc ← ZeroExtend64(PC);
offset ← SignExtend16(s) << 2;
label ← ZeroExtend64((pc + offset) + 1);
IF (MalformedAddress(label))
   THROW IADDERR, label;
TRa ← Register(label);
```

Description:

This instruction calculates a target address by adding a constant value onto the PC of the current instruction. The constant is formed by taking the sign-extended 16-bit immediate s, shifting it left by 2 bits and adding 1. If the computed target address is outside the implemented effective address range an IADDERR exception is generated. Otherwise, the target address is placed in the target register TR_a.

The encoding contains a single bit, labeled l, which is used to indicate whether it is likely (1) or unlikely (0) that control will flow to that target address. This bit is encoded as 1 if the instruction mnemonic is 'PTA' or 'PTA/L', or as 0 if the mnemonic is 'PTA/U'.

Possible exceptions:

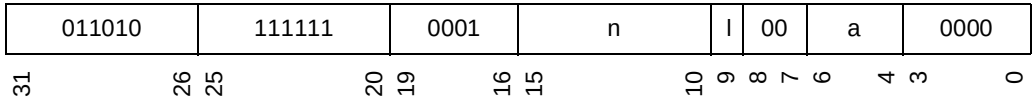
IADDERR

Notes:

The 'label' in the assembly syntax represents the absolute address of the target instruction with bit 0 set to 1 to indicate SHmedia mode.

PTABS Rn, TRa

PTABS Rn, TRA




```
address ← ZeroExtend64(Rn);
target ← address;
IF (MalformedAddress(target) OR ((target ∧ 0x3) = 0x3))
   THROW IADDERR, target;
TRa ← Register(target);
```

Description:

This instruction uses a target address specified by R_n. If the target address is outside the implemented effective address range an IADDERR exception is generated. If the target address indicates a misaligned SHmedia instruction an IADDERR exception is generated. Otherwise, the target address is placed in the target register TR_a.

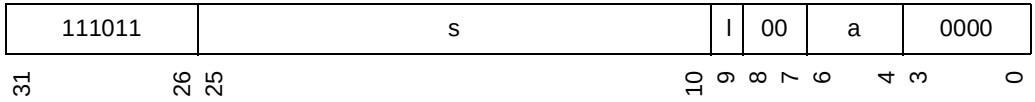
The encoding contains a single bit, labeled l, which is used to indicate whether it is likely (1) or unlikely (0) that control will flow to that target address. This bit is encoded as 1 if the instruction mnemonic is 'PTABS' or 'PTABS/L', or as 0 if the mnemonic is 'PTABS/U'.

Possible exceptions:

IADDERR

PTB label, TRa

PTB label, TRa




```
pc ← ZeroExtend64(PC);
offset ← SignExtend16(s) << 2;
label ← ZeroExtend64(pc + offset);
IF (MalformedAddress(label))
   THROW IADDERR, label;
TRa ← Register(label);
```

Description:

This instruction calculates a target address by adding a constant value onto the PC of the current instruction. The constant is formed by taking the sign-extended 16-bit immediate s and shifting it left by 2 bits. If the computed target address is outside the implemented effective address range an IADDERR exception is generated. Otherwise, the target address is placed in the target register TR_a.

The encoding contains a single bit, labeled l, which is used to indicate whether it is likely (1) or unlikely (0) that control will flow to that target address. This bit is encoded as 1 if the instruction mnemonic is 'PTB' or 'PTB/L', or as 0 if the mnemonic is 'PTB/U'.

Possible exceptions:

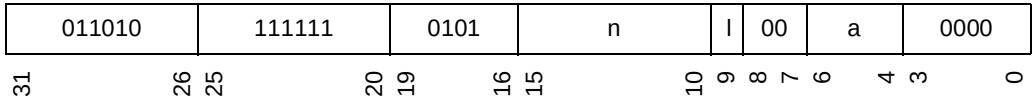
IADDERR

Notes:

The 'label' in the assembly syntax represents the absolute address of the target instruction with bit 0 set to 0 to indicate SHcompact mode.

PTREL Rn, TRa

PTREL Rn, TRa




```
pc ← ZeroExtend64(PC);
source ← SignExtend64(Rn);
target ← ZeroExtend64(pc + source);
IF (MalformedAddress(target) OR ((target ∧ 0x3) = 0x3))
   THROW IADDERR, target;
TRa ← Register(target);
```

Description:

This instruction calculates a target address by adding R_n onto the PC of the current instruction. If the computed target address is outside the implemented effective address range an IADDERR exception is generated.If the target address indicates a misaligned SHmedia instruction an IADDERR exception is generated. Otherwise, the target address is placed in the target register TR_a.

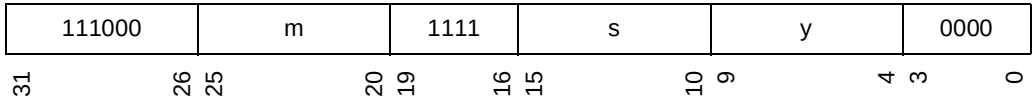
The encoding contains a single bit, labeled l, which is used to indicate whether it is likely (1) or unlikely (0) that control will flow to that target address. This bit is encoded as 1 if the instruction mnemonic is 'PTREL' or 'PTREL/L', or as 0 if the mnemonic is 'PTREL/U'.

Possible exceptions:

IADDERR

PUTCFG Rm, disp, Ry

PUTCFG Rm, disp, Ry




```
md ← ZeroExtend1(MD);
base ← ZeroExtend64(Rm);
disp ← SignExtend6(s);
value ← ZeroExtend64(Ry);
index ← ZeroExtend64(base + disp);
IF (md = 0)
   THROW RESINST;
IF (IsUndefinedConfigurationRegister(index))
   UNDEFINED();
WriteConfigurationRegister(index, value);
```

Description:

This instruction copies R_y to a configuration register. The destination configuration register is identified by adding R_m to the sign-extended 6-bit immediate s.

PUTCFG is a privileged instruction.

A write to an undefined configuration register results in architecturally-undefined behavior. Note that configuration registers do not, in general, have simple read/write semantics.

Possible exceptions:

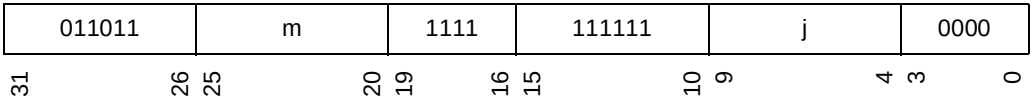
RESINST

Notes:

The ‘disp’ in the assembly syntax represents the immediate s after sign extension.

PUTCON Rm, CRj

PUTCON Rm, CRj




```
md ← ZeroExtend1(MD);
value ← ZeroExtend64(Rm);
index ← ZeroExtend6(j);
IF ((md = 0) AND IsPrivilegedControlRegister(index))
   THROW RESINST;
IF (IsUndefinedControlRegister(index))
   UNDEFINED();
WriteControlRegister(index, value);
```

Description:

This instruction copies R_m to CR_j.

PUTCON to a privileged control register is a privileged instruction. PUTCON to a user-accessible control register is not a privileged instruction.

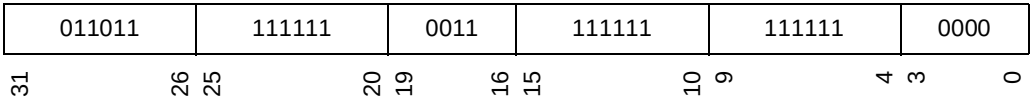
A write to an undefined control register results in architecturally-undefined behavior. Note that control registers do not, in general, have simple read/write semantics.

Possible exceptions:

RESINST

RTE

RTE




```
ssr ← ZeroExtend64(SSR);
spc ← ZeroExtend64(SPC);
pssr ← ZeroExtend64(PSSR);
pspc ← ZeroExtend64(PSPC);
md ← ZeroExtend1(MD);
IF (md = 0)
   THROW RESINST;
IF (IsInvalidPC(spc) OR IsInvalidSR(ssr))
   UNDEFINED();
sr ← ssr;
isa ← spc ∧ 0x1;
newpc ← spc ∧ (~ 0x1);
ssr ← pssr;
spc ← pspc;
SSR ← Register(ssr);
SPC ← Register(spc);
ISA ← Bit(isa);
SR ← Register(sr);
PC' ← Register(newpc);
```

Description:

This instruction restores the PC and the ISA from the saved value held in SPC, and restores the SR from the saved value held in SSR. It then restores SPC from PSPC, and SSR from PSSR. If SPC is an inappropriate value for the program counter, or if SSR is an inappropriate value for the status register, the behavior is architecturally undefined.

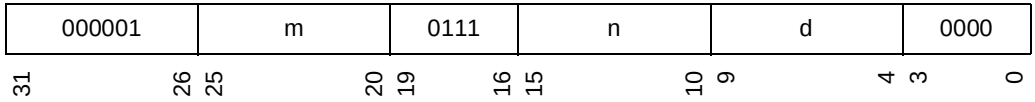
This is a privileged instruction.

Possible exceptions:

RESINST

SHARD Rm, Rn, Rd

SHARD Rm, Rn, Rd



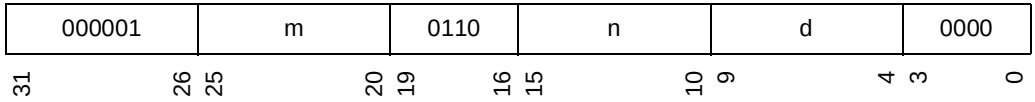
source1 ← SignExtend$_{64}$(R_m);
source2 ← ZeroExtend$_6$(R_n);
result ← source1 >> source2;
R_d ← Register(result);

Description:

This instruction performs an arithmetic right shift of R_m by a shift amount specified in the lowest 6 bits of R_n and places the result in R_d. The highest 58 bits of R_n are ignored.

SHARD.L Rm, Rn, Rd

SHARD.L Rm, Rn, Rd



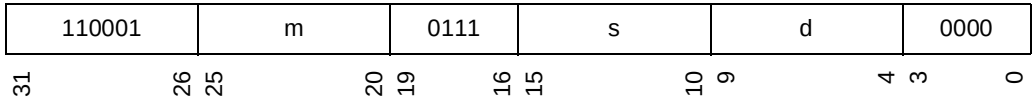

```
source1 ← SignExtend32(Rm);
source2 ← ZeroExtend5(Rn);
result ← SignExtend32(source1 >> source2);
Rd ← Register(result);
```

Description:

This instruction performs an arithmetic right shift on the lowest 32 bits of R_m by a shift amount specified in the lowest 5 bits of R_n and places the sign-extended 32-bit result in R_d. The highest 59 bits of R_n are ignored.

SHARI Rm, imm, Rd

SHARI Rm, imm, Rd



source1 ← $SignExtend_{64}(R_m)$;
imm ← $ZeroExtend_6(SignExtend_6(s))$;
result ← source1 >> imm;
R_d ← Register(result);

Description:

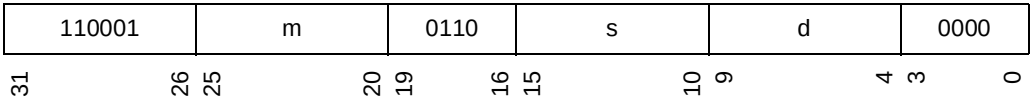
This instruction performs an arithmetic right shift of R_m by a shift amount specified in the 6-bit immediate s and places the result in R_d.

Notes:

The ‘imm’ in the assembly syntax represents the immediate s after zero extension.

SHARI.L Rm, imm, Rd

SHARI.L Rm, imm, Rd




```
source1 ← SignExtend32(Rm);
imm ← ZeroExtend5(SignExtend6(s));
result ← SignExtend32(source1 >> imm);
Rd ← Register(result);
```

Description:

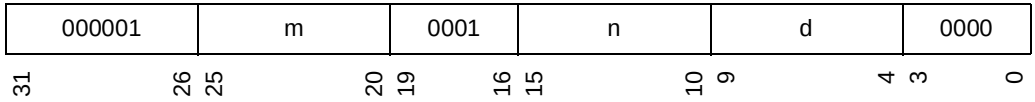
This instruction performs an arithmetic right shift on the lowest 32 bits of R_m by a shift amount specified in the lowest 5 bits of the 6-bit immediate s and places the sign-extended 32-bit result in R_d. The highest bit of the 6-bit immediate s is ignored.

Notes:

The 'imm' in the assembly syntax represents the immediate s after zero extension.

SHLLD Rm, Rn, Rd

SHLLD Rm, Rn, Rd



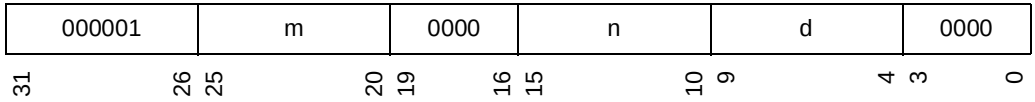

```
source1 ← ZeroExtend64(Rm);
source2 ← ZeroExtend6(Rn);
result ← source1 << source2;
Rd ← Register(result);
```

Description:

This instruction performs a logical left shift of R_m by a shift amount specified in the lowest 6 bits of R_n and places the result in R_d. The highest 58 bits of R_n are ignored.

SHLLD.L Rm, Rn, Rd

SHLLD.L Rm, Rn, Rd



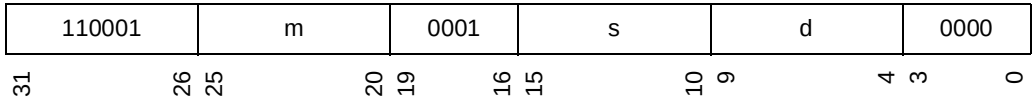
source1 ← ZeroExtend$_{32}$(R_m);
source2 ← ZeroExtend$_5$(R_n);
result ← SignExtend$_{32}$(source1 << source2);
R_d ← Register(result);

Description:

This instruction performs a logical left shift on the lowest 32 bits of R_m by a shift amount specified in the lowest 5 bits of R_n and places the sign-extended 32-bit result in R_d. The highest 59 bits of R_n are ignored.

SHLLI Rm, imm, Rd

SHLLI Rm, imm, Rd



source1 ← ZeroExtend$_{64}$(R_m);
imm ← ZeroExtend$_6$(SignExtend$_6$(s));
result ← source1 << imm;
R_d ← Register(result);

Description:

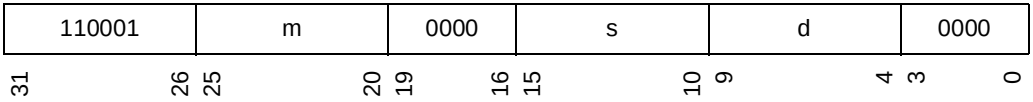
This instruction performs a logical left shift of R_m by a shift amount specified in the 6-bit immediate s and places the result in R_d.

Notes:

The 'imm' in the assembly syntax represents the immediate s after zero extension.

SHLLI.L Rm, imm, Rd

SHLLI.L Rm, imm, Rd



source1 ← ZeroExtend$_{32}$(R_m);
imm ← ZeroExtend$_5$(SignExtend$_6$(s));
result ← SignExtend$_{32}$(source1 << imm);
R_d ← Register(result);

Description:

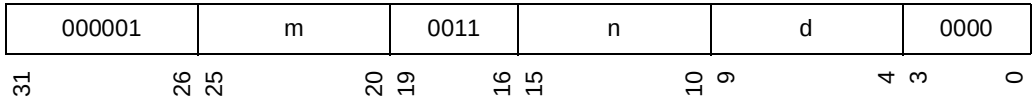
This instruction performs a logical left shift on the lowest 32 bits of R_m by a shift amount specified in the lowest 5 bits of the 6-bit immediate s and places the sign-extended 32-bit result in R_d. The highest bit of the 6-bit immediate s is ignored.

Notes:

The 'imm' in the assembly syntax represents the immediate s after zero extension.

SHLRD Rm, Rn, Rd

SHLRD Rm, Rn, Rd



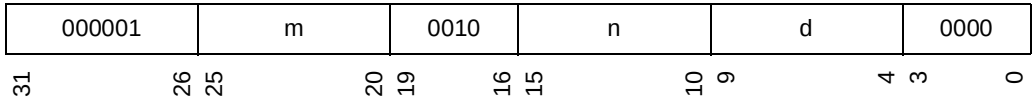

```
source1 ← ZeroExtend64(Rm);
source2 ← ZeroExtend6(Rn);
result ← source1 >> source2;
Rd ← Register(result);
```

Description:

This instruction performs a logical right shift of R_m by a shift amount specified in the lowest 6 bits of R_n and places the result in R_d. The highest 58 bits of R_n are ignored.

SHLRD.L Rm, Rn, Rd

SHLRD.L Rm, Rn, Rd



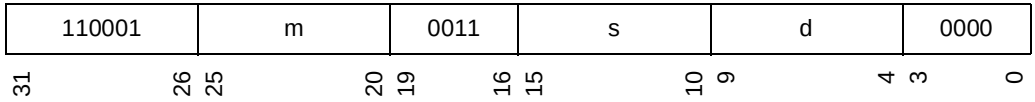
source1 ← ZeroExtend$_{32}$(R_m);
source2 ← ZeroExtend$_5$(R_n);
result ← SignExtend$_{32}$(source1 >> source2);
R_d ← Register(result);

Description:

This instruction performs a logical right shift on the lowest 32 bits of R_m by a shift amount specified in the lowest 5 bits of R_n and places the sign-extended 32-bit result in R_d. The highest 59 bits of R_n are ignored.

SHLRI Rm, imm, Rd

SHLRI Rm, imm, Rd



source1 ← ZeroExtend$_{64}$(R_m);
imm ← ZeroExtend$_6$(SignExtend$_6$(s));
result ← source1 >> imm;
R_d ← Register(result);

Description:

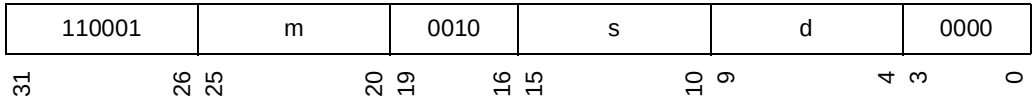
This instruction performs a logical right shift of R_m by a shift amount specified in the 6-bit immediate s and places the result in R_d.

Notes:

The ‘imm’ in the assembly syntax represents the immediate s after zero extension.

SHLRI.L Rm, imm, Rd

SHLRI.L Rm, imm, Rd




```
source1 ← ZeroExtend32(Rm);
imm ← ZeroExtend5(SignExtend6(s));
result ← SignExtend32(source1 >> imm);
Rd ← Register(result);
```

Description:

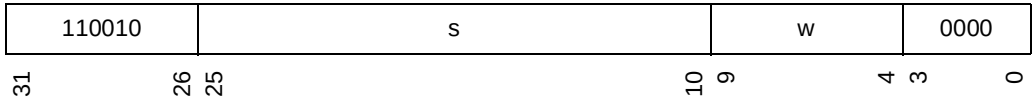
This instruction performs a logical right shift on the lowest 32 bits of R_m by a shift amount specified in the lowest 5 bits of the 6-bit immediate s and places the sign-extended 32-bit result in R_d. The highest bit of the 6-bit immediate s is ignored.

Notes:

The 'imm' in the assembly syntax represents the immediate s after zero extension.

SHORI imm, Rw

SHORI s, Rw




```
imm ← ZeroExtend16(SignExtend16(s));
source2_result ← ZeroExtend64(Rw);
source2_result ← (source2_result << 16) ∨ imm;
Rw ← Register(source2_result);
```

Description:

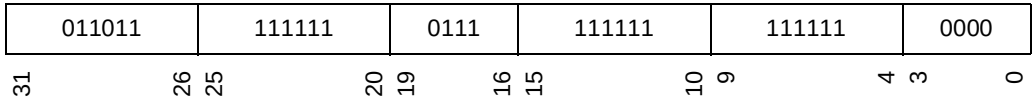
This instruction left shifts R_w by 16, performs a bitwise OR with the 16-bit immediate field s, and places the result in R_w.

Notes:

The 'imm' in the assembly syntax represents the immediate s after zero extension.

SLEEP

SLEEP




```
md ← ZeroExtend1(MD);
IF (md = 0)
   THROW RESINST;
SLEEP();
```

Description:

This instruction places the CPU into sleep mode. Execution of instructions is stopped, though the state of the CPU is preserved. Sleep mode is exited when an asynchronous event (an interrupt or a reset) arrives, and then instruction execution continues. If the event causes an event handler to be launched, execution continues with that handler, otherwise execution continues with the next instruction after the SLEEP instruction.

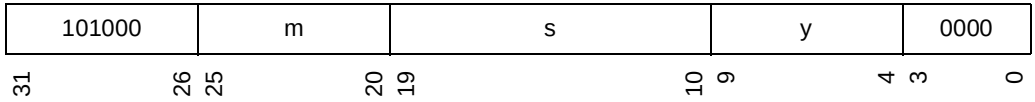
This is a privileged instruction.

Possible exceptions:

RESINST

ST.B Rm, disp, Ry

ST.B Rm, disp, Ry




```
base ← ZeroExtend64(Rm);
disp ← SignExtend10(s);
value ← ZeroExtend8(Ry);
address ← ZeroExtend64(base + disp);
WriteMemory8(address, value);
```

Description:

This instruction stores a byte to the effective address formed by adding R_m to the sign-extended 10-bit immediate s. The byte to be stored is held in the lowest 8 bits of R_y. In exceptional cases, an appropriate exception is raised.

Possible exceptions:

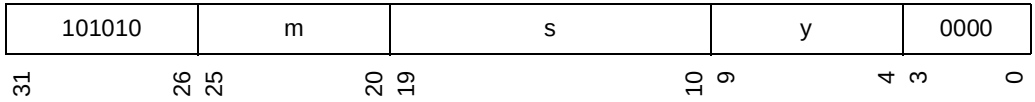
WADDERR, WTLBMISS, WRITEPROT

Notes:

The ‘disp’ in the assembly syntax represents the immediate s after sign extension.

ST.L Rm, disp, Ry

ST.L Rm, disp, Ry




```
base ← ZeroExtend64(Rm);
disp ← SignExtend10(s) << 2;
value ← ZeroExtend32(Ry);
address ← ZeroExtend64(base + disp);
WriteMemory32(address, value);
```

Description:

This instruction stores a long-word to the effective address formed by adding R_m to the sign-extended 10-bit immediate s multiplied by 4. The long-word to be stored is held in the lowest 32 bits of R_y. In exceptional cases, including misaligned stores, an appropriate exception is raised.

Possible exceptions:

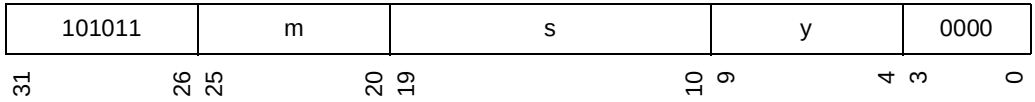
WADDERR, WTLBMISS, WRITEPROT

Notes:

The ‘disp’ in the assembly syntax represents the immediate s after sign extension and scaling.

ST.Q Rm, disp, Ry

ST.Q Rm, disp, Ry




```
base ← ZeroExtend64(Rm);
disp ← SignExtend10(s) << 3;
value ← ZeroExtend64(Ry);
address ← ZeroExtend64(base + disp);
WriteMemory64(address, value);
```

Description:

This instruction stores a quad-word to the effective address formed by adding R_m to the sign-extended 10-bit immediate s multiplied by 8. The quad-word to be stored is held in R_y. In exceptional cases, including misaligned stores, an appropriate exception is raised.

Possible exceptions:

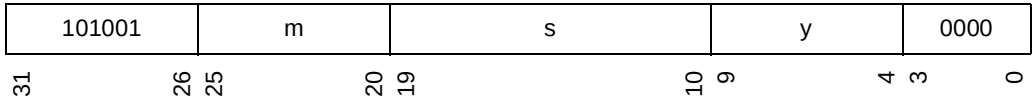
WADDERR, WTLBMISS, WRITEPROT

Notes:

The ‘disp’ in the assembly syntax represents the immediate s after sign extension and scaling.

ST.W Rm, disp, Ry

ST.W Rm, disp, Ry




```
base ← ZeroExtend64(Rm);
disp ← SignExtend10(s) << 1;
value ← ZeroExtend16(Ry);
address ← ZeroExtend64(base + disp);
WriteMemory16(address, value);
```

Description:

This instruction stores a word to the effective address formed by adding R_m to the sign-extended 10-bit immediate s multiplied by 2. The word to be stored is held in the lowest 16 bits of R_y. In exceptional cases, including misaligned stores, an appropriate exception is raised.

Possible exceptions:

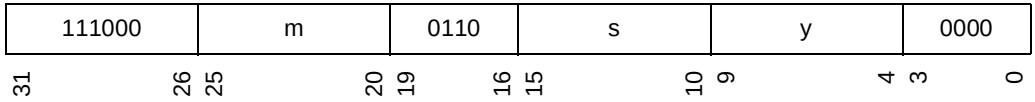
WADDERR, WTLBMISS, WRITEPROT

Notes:

The ‘disp’ in the assembly syntax represents the immediate s after sign extension and scaling.

STHI.L Rm, disp, Ry

STHI.L Rm, disp, Ry




```
base ← ZeroExtend64(Rm);
disp ← SignExtend6(s);
value ← ZeroExtend32(Ry);
address ← base + disp;
bytecount ← (address ∧ 0x3) + 1;
bitcount ← bytecount × 8;
IF (IsLittleEndian())
   start ← 32 - bitcount;
ELSE
   start ← 0;
WriteMemoryHighbitcount(address, ZeroExtendbitcount(value >> start));
```

Description:

This instruction stores the high part of a misaligned long-word from R_y into memory. The effective address is formed by adding the sign-extended 6-bit immediate s to R_m. The effective address points to the highest byte in the misaligned long-word. The address of the lowest byte in the high part of the misaligned long-word is determined by masking the least significant 2 bits of the effective address to 0.

This instruction stores into the inclusive range of memory bytes starting at that lowest byte and ending at that highest byte. If the effective address is actually 4-byte aligned, then all 4 bytes are stored. The stored bytes are taken from appropriate bytes within R_y and any other bytes are ignored.

This instruction can be used in conjunction with STLO.L to store a misaligned long-word from a register into memory. In this case, the STHI.L effective address should be 3 bytes larger than the STLO.L effective address.

Possible exceptions:

WADDERR, WTLBMISS, WRITEPROT

STHI.L byte mappings:

The mapping between byte locations in memory and byte positions in the destination register is shown below for each value of the low 2 bits in the effective address (EA). Each byte in the register is either ignored or maps to the given memory address.

Little endian mode

Low 2 bits of EA↓	Bit 63 / Target register / Bit 0							
	Byte 7	Byte 6	Byte 5	Byte 4	Byte 3	Byte 2	Byte 1	Byte 0
0x0	ignore				EA-0	ignore		
0x1	ignore				EA-0	EA-1	ignore	
0x2	ignore				EA-0	EA-1	EA-2	ignore
0x3	ignore				EA-0	EA-1	EA-2	EA-3

Big endian mode

Low 2 bits of EA↓	Bit 63 / Target register / Bit 0							
	Byte 0	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7
0x0	ignore							EA-0
0x1	ignore						EA-1	EA-0
0x2	ignore					EA-2	EA-1	EA-0
0x3	ignore				EA-3	EA-2	EA-1	EA-0

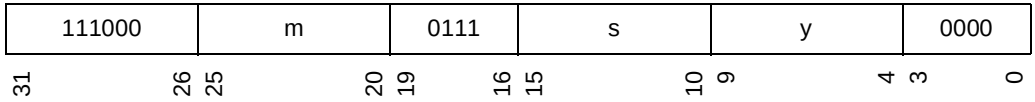
Notes:

The ‘disp’ in the assembly syntax represents the immediate s after sign extension.

When the memory access for STHI.L causes an exception, the TEA control register is initialized with the effective address of the access. This corresponds to the address of the highest byte in the misaligned long-word.

STHI.Q Rm, disp, Ry

STHI.Q Rm, disp, Ry




```
base ← ZeroExtend64(Rm);
disp ← SignExtend6(s);
value ← ZeroExtend64(Ry);
address ← base + disp;
bytecount ← (address ∧ 0x7) + 1;
bitcount ← bytecount × 8;
IF (IsLittleEndian())
   start ← 64 - bitcount;
ELSE
   start ← 0;
WriteMemoryHighbitcount(address, ZeroExtendbitcount(value >> start));
```

Description:

This instruction stores the high part of a misaligned quad-word from R_y into memory. The effective address is formed by adding the sign-extended 6-bit immediate s to R_m. The effective address points to the highest byte in the misaligned quad-word. The address of the lowest byte in the high part of the misaligned quad-word is determined by masking the least significant 3 bits of the effective address to 0.

This instruction stores into the inclusive range of memory bytes starting at that lowest byte and ending at that highest byte. If the effective address is actually 8-byte aligned, then all 8 bytes are stored. The stored bytes are taken from appropriate bytes within R_y and any other bytes are ignored.

This instruction can be used in conjunction with STLO.Q to store a misaligned quad-word from a register into memory. In this case, the STHI.Q effective address should be 7 bytes larger than the STLO.Q effective address.

Possible exceptions:

WADDERR, WTLBMISS, WRITEPROT

STHI.Q byte mappings:

The mapping between byte locations in memory and byte positions in the destination register is shown below for each value of the low 3 bits in the effective address (EA). Each byte in the register is either ignored or maps to the given memory address.

Little endian mode

Low 3 bits of EA↓	Bit 63 ← Target register → Bit 0							
	Byte 7	**Byte 6**	**Byte 5**	**Byte 4**	**Byte 3**	**Byte 2**	**Byte 1**	**Byte 0**
0x0	EA-0	ignore						
0x1	EA-0	EA-1	ignore					
0x2	EA-0	EA-1	EA-2	ignore				
0x3	EA-0	EA-1	EA-2	EA-3	ignore			
0x4	EA-0	EA-1	EA-2	EA-3	EA-4	ignore		
0x5	EA-0	EA-1	EA-2	EA-3	EA-4	EA-5	ignore	
0x6	EA-0	EA-1	EA-2	EA-3	EA-4	EA-5	EA-6	ignore
0x7	EA-0	EA-1	EA-2	EA-3	EA-4	EA-5	EA-6	EA-7

Big endian mode

Low 3 bits of EA↓	Bit 63 ← Target register → Bit 0							
	Byte 0	**Byte 1**	**Byte 2**	**Byte 3**	**Byte 4**	**Byte 5**	**Byte 6**	**Byte 7**
0x0	ignore							EA-0
0x1	ignore						EA-1	EA-0
0x2	ignore					EA-2	EA-1	EA-0
0x3	ignore				EA-3	EA-2	EA-1	EA-0
0x4	ignore			EA-4	EA-3	EA-2	EA-1	EA-0

Big endian mode	Bit 63	Target register					Bit 0
Low 3 bits of EA↓	**Byte 0**	**Byte 1**	**Byte 2**	**Byte 3**	**Byte 4**	**Byte 5**	**Byte 6**	**Byte 7**
0x5	ignore		EA-5	EA-4	EA-3	EA-2	EA-1	EA-0
0x6	ignore	EA-6	EA-5	EA-4	EA-3	EA-2	EA-1	EA-0
0x7	EA-7	EA-6	EA-5	EA-4	EA-3	EA-2	EA-1	EA-0

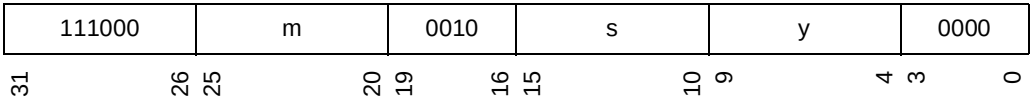
Notes:

The ‘disp’ in the assembly syntax represents the immediate s after sign extension.

When the memory access for STHI.Q causes an exception, the TEA control register is initialized with the effective address of the access. This corresponds to the address of the highest byte in the misaligned quad-word.

STLO.L Rm, disp, Ry

STLO.L Rm, disp, Ry




```
base ← ZeroExtend64(Rm);
disp ← SignExtend6(s);
value ← ZeroExtend32(Ry);
address ← ZeroExtend64(base + disp);
bytecount ← 4 - (address ∧ 0x3);
bitcount ← bytecount × 8;
IF (IsLittleEndian())
   start ← 0;
ELSE
   start ← 32 - bitcount;
WriteMemoryLowbitcount(address, ZeroExtendbitcount(value >> start));
```

Description:

This instruction stores the low part of a misaligned long-word from R_y into memory. The effective address is formed by adding the sign-extended 6-bit immediate s to R_m. The effective address points to the lowest byte in the misaligned long-word. The address of the highest byte in the low part of the misaligned long-word is determined by setting the least significant 2 bits of the effective address to 1.

This instruction stores into the inclusive range of memory bytes starting at that lowest byte and ending at that highest byte. If the effective address is actually 4-byte aligned, then all 4 bytes are stored. The stored bytes are taken from appropriate bytes within R_y and any other bytes are ignored.

This instruction can be used in conjunction with STHI.L to store a misaligned long-word from a register into memory. In this case, the STHI.L effective address should be 3 bytes larger than the STLO.L effective address.

Possible exceptions:

WADDERR, WTLBMISS, WRITEPROT

STLO.L Byte Mappings:

The mapping between byte locations in memory and byte positions in the destination register is shown below for each value of the low 2 bits in the effective address (EA). Each byte in the register is either ignored or maps to the given memory address.

Little endian mode

Low 2 bits of EA↓	Bit 63 ... Target register ... Bit 0							
	Byte 7	**Byte 6**	**Byte 5**	**Byte 4**	**Byte 3**	**Byte 2**	**Byte 1**	**Byte 0**
0x0	ignore				EA+3	EA+2	EA+1	EA+0
0x1	ignore					EA+2	EA+1	EA+0
0x2	ignore						EA+1	EA+0
0x3	ignore							EA+0

Big endian mode

Low 2 bits of EA↓	Bit 63 ... Target register ... Bit 0							
	Byte 0	**Byte 1**	**Byte 2**	**Byte 3**	**Byte 4**	**Byte 5**	**Byte 6**	**Byte 7**
0x0	ignore				EA+0	EA+1	EA+2	EA+3
0x1	ignore				EA+0	EA+1	EA+2	ignore
0x2	ignore				EA+0	EA+1	ignore	
0x3	ignore				EA+0	ignore		

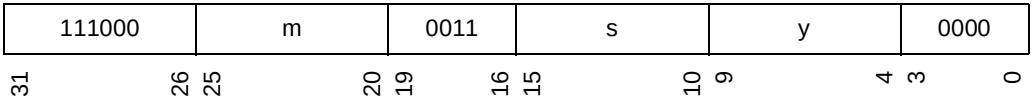
Notes:

The ‘disp’ in the assembly syntax represents the immediate s after sign extension.

When the memory access for STLO.L causes an exception, the TEA control register is initialized with the effective address of the access. This corresponds to the address of the lowest byte in the misaligned long-word.

STLO.Q Rm, disp, Ry

STLO.Q Rm, disp, Ry




```
base ← ZeroExtend64(Rm);
disp ← SignExtend6(s);
value ← ZeroExtend64(Ry);
address ← ZeroExtend64(base + disp);
bytecount ← 8 - (address ∧ 0x7);
bitcount ← bytecount × 8;
IF (IsLittleEndian())
   start ← 0;
ELSE
   start ← 64 - bitcount;
WriteMemoryLowbitcount(address, ZeroExtendbitcount(value >> start));
```

Description:

This instruction stores the low part of a misaligned quad-word from R_y into memory. The effective address is formed by adding the sign-extended 6-bit immediate s to R_m. The effective address points to the lowest byte in the misaligned quad-word. The address of the highest byte in the low part of the misaligned quad-word is determined by setting the least significant 3 bits of the effective address to 1.

This instruction stores into the inclusive range of memory bytes starting at that lowest byte and ending at that highest byte. If the effective address is actually 8-byte aligned, then all 8 bytes are stored. The stored bytes are taken from appropriate bytes within R_y and any other bytes are ignored.

This instruction can be used in conjunction with STHI.Q to store a misaligned quad-word from a register into memory. In this case, the STHI.Q effective address should be 7 bytes larger than the STLO.Q effective address.

Possible exceptions:

WADDERR, WTLBMISS, WRITEPROT

STLO.Q Byte Mappings:

The mapping between byte locations in memory and byte positions in the destination register is shown below for each value of the low 3 bits in the effective address (EA). Each byte in the register is either ignored or maps to the given memory address.

Little endian mode

Low 3 bits of EA↓	Bit 63 — Target register — Bit 0							
	Byte 7	**Byte 6**	**Byte 5**	**Byte 4**	**Byte 3**	**Byte 2**	**Byte 1**	**Byte 0**
0x0	EA+7	EA+6	EA+5	EA+4	EA+3	EA+2	EA+1	EA+0
0x1	ignore	EA+6	EA+5	EA+4	EA+3	EA+2	EA+1	EA+0
0x2	ignore		EA+5	EA+4	EA+3	EA+2	EA+1	EA+0
0x3	ignore			EA+4	EA+3	EA+2	EA+1	EA+0
0x4	ignore				EA+3	EA+2	EA+1	EA+0
0x5	ignore					EA+2	EA+1	EA+0
0x6	ignore						EA+1	EA+0
0x7	ignore							EA+0

Big endian mode

Low 3 bits of EA↓	Bit 63 — Target register — Bit 0							
	Byte 0	**Byte 1**	**Byte 2**	**Byte 3**	**Byte 4**	**Byte 5**	**Byte 6**	**Byte 7**
0x0	EA+0	EA+1	EA+2	EA+3	EA+4	EA+5	EA+6	EA+7
0x1	EA+0	EA+1	EA+2	EA+3	EA+4	EA+5	EA+6	ignore
0x2	EA+0	EA+1	EA+2	EA+3	EA+4	EA+5	ignore	
0x3	EA+0	EA+1	EA+2	EA+3	EA+4	ignore		
0x4	EA+0	EA+1	EA+2	EA+3	ignore			

Big endian mode	Bit 63	Target register					Bit 0
Low 3 bits of EA↓	Byte 0	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7
0x5	EA+0	EA+1	EA+2	ignore				
0x6	EA+0	EA+1	ignore					
0x7	EA+0	ignore						

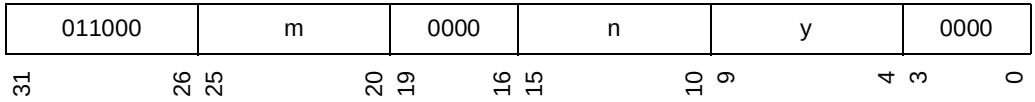
Notes:

The ‘disp’ in the assembly syntax represents the immediate s after sign extension.

When the memory access for STLO.Q causes an exception, the TEA control register is initialized with the effective address of the access. This corresponds to the address of the lowest byte in the misaligned quad-word.

STX.B Rm, Rn, Ry

STX.B Rm, Rn, Ry




```
base ← ZeroExtend64(Rm);
index ← SignExtend64(Rn);
value ← ZeroExtend8(Ry);
address ← ZeroExtend64(base + index);
WriteMemory8(address, value);
```

Description:

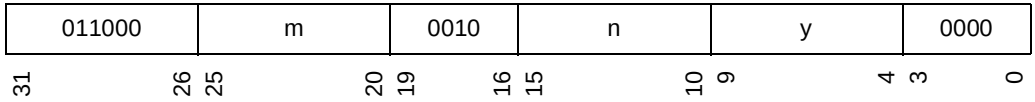
This instruction stores a byte to the effective address formed by adding R_m and R_n. The byte to be stored is held in the lowest 8 bits of R_y. In exceptional cases, an appropriate exception is raised.

Possible exceptions:

WADDERR, WTLBMISS, WRITEPROT

STX.L Rm, Rn, Ry

STX.L Rm, Rn, Ry




```
base ← ZeroExtend64(Rm);
index ← SignExtend64(Rn);
value ← ZeroExtend32(Ry);
address ← ZeroExtend64(base + index);
WriteMemory32(address, value);
```

Description:

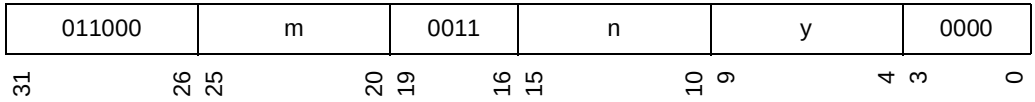
This instruction stores a long-word to the effective address formed by adding R_m and R_n. The long-word to be stored is held in the lowest 32 bits of R_y. In exceptional cases, including misaligned stores, an appropriate exception is raised.

Possible exceptions:

WADDERR, WTLBMISS, WRITEPROT

STX.Q Rm, Rn, Ry

STX.Q Rm, Rn, Ry




```
base ← ZeroExtend64(Rm);
index ← SignExtend64(Rn);
value ← ZeroExtend64(Ry);
address ← ZeroExtend64(base + index);
WriteMemory64(address, value);
```

Description:

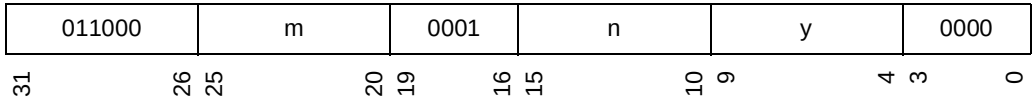
This instruction stores a quad-word to the effective address formed by adding R_m and R_n. The quad-word to be stored is held in R_y. In exceptional cases, including misaligned stores, an appropriate exception is raised.

Possible exceptions:

WADDERR, WTLBMISS, WRITEPROT

STX.W Rm, Rn, Ry

STX.W Rm, Rn, Ry




```
base ← ZeroExtend64(Rm);
index ← SignExtend64(Rn);
value ← ZeroExtend16(Ry);
address ← ZeroExtend64(base + index);
WriteMemory16(address, value);
```

Description:

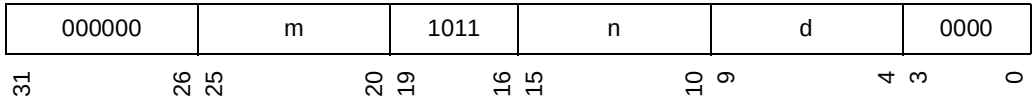
This instruction stores a word to the effective address formed by adding R_m and R_n. The word to be stored is held in the lowest 16 bits of R_y. In exceptional cases, including misaligned stores, an appropriate exception is raised.

Possible exceptions:

WADDERR, WTLBMISS, WRITEPROT

SUB Rm, Rn, Rd

SUB Rm, Rn, Rd



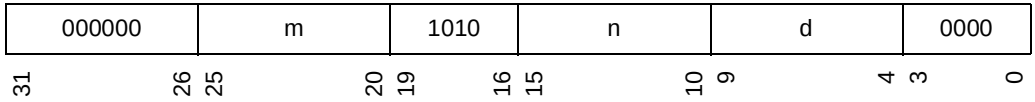
source1 ← SignExtend$_{64}$(R_m);
source2 ← SignExtend$_{64}$(R_n);
result ← source1 - source2;
R_d ← Register(result);

Description:

This instruction subtracts R_n from R_m and places the result in R_d.

SUB.L Rm, Rn, Rd

SUB.L Rm, Rn, Rd



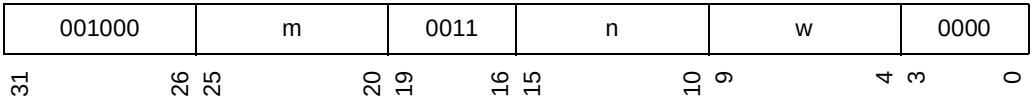

```
source1 ← SignExtend32(Rm);
source2 ← SignExtend32(Rn);
result ← SignExtend32(source1 - source2);
Rd ← Register(result);
```

Description:

This instruction subtracts the lowest 32 bits of R_n from the lowest 32 bits of R_m and places the sign-extended 32-bit value of the result in R_d. The highest 32 bits of R_m and the highest 32 bits of R_n are ignored.

SWAP.Q Rm, Rn, Rw

SWAP.Q Rm, Rn, Rw




```
base ← ZeroExtend64(Rm);
index ← SignExtend64(Rn);
result_value ← ZeroExtend64(Rw);
address ← ZeroExtend64(base + index);
result_value ← ZeroExtend64(SwapMemory64(address, result_value));
Rw ← Register(result_value);
```

Description:

This instruction is an atomic read-modify-write operation on the 64-bit memory object at the effective address formed by adding R_m and R_n. The quad-word held in R_w is written to the effective address, and the previous quad-word contents are returned in R_w.

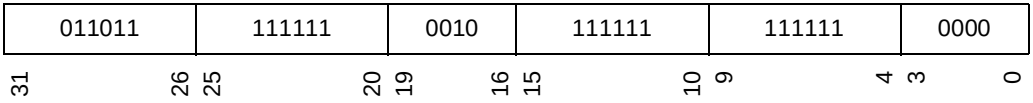
The memory system guarantees that the read and write parts of the swap instruction are implemented atomically on the target memory location. Only swaps to 8-byte aligned addresses are allowed. In exceptional cases, including misaligned swaps, an appropriate exception is raised.

Possible exceptions:

WADDERR, WTLBMISS, READPROT, WRITEPROT

SYNCI

SYNCI




```
SYNCI();
```

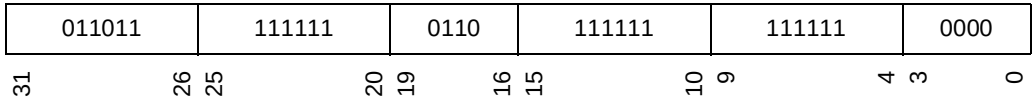
Description:

This instruction is used to synchronize instruction fetch. Execution of a SYNCI ensures that all previous instructions are completed before any subsequent instruction is fetched.

Further information on SYNCI and examples of usage can be found in *Volume 1, Chapter 6: SHmedia memory instructions*.

SYNCO

SYNCO




```
SYNCO();
```

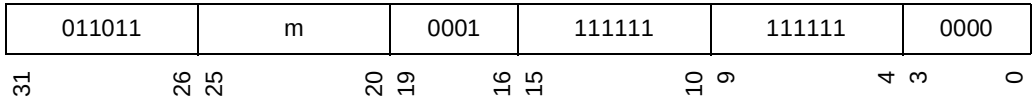
Description:

This instruction is used to synchronize data operations. Execution of a SYNCO ensures that all data operations from previous instructions are completed before any data operations from subsequent instructions are started.

Further information on SYNCO and examples of usage can be found in *Volume 1, Chapter 6: SHmedia memory instructions*.

TRAPA Rm

TRAPA Rm




```
tra ← ZeroExtend64(Rm);
THROW TRAP, tra;
```

Description:

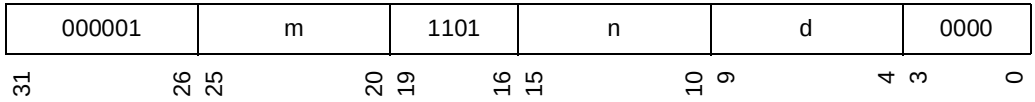
This instruction causes a pre-execution trap. The value of R_m is used by the handler launch sequence to characterize the trap.

Possible exceptions:

TRAP

XOR Rm, Rn, Rd

XOR Rm, Rn, Rd



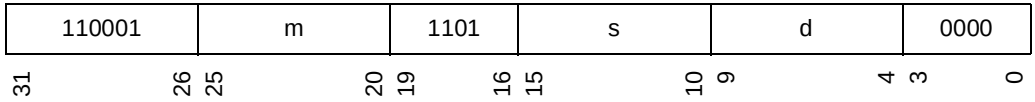

```
source1 ← SignExtend64(Rm);
source2 ← SignExtend64(Rn);
result ← source1 ⊕ source2;
Rd ← Register(result);
```

Description:

This instruction performs a bitwise XOR of R_m with R_n and places the result in R_d.

XORI Rm, imm, Rd

XORI Rm, imm, Rd



source1 ← SignExtend$_{64}$(R_m);
imm ← SignExtend$_6$(s);
result ← source1 ⊕ imm;
R_d ← Register(result);

Description:

This instruction performs a bitwise XOR of R_m with the sign-extended 6-bit immediate s and places the result in R_d.

Notes:

The 'imm' in the assembly syntax represents the immediate s after sign extension.

SuperH

A

SHmedia instruction encoding

A.1 Major formats

The major formats can be quickly distinguished by decoding the 6-bit opcode. Opcodes labeled ‘reserved’ are not associated with any format.

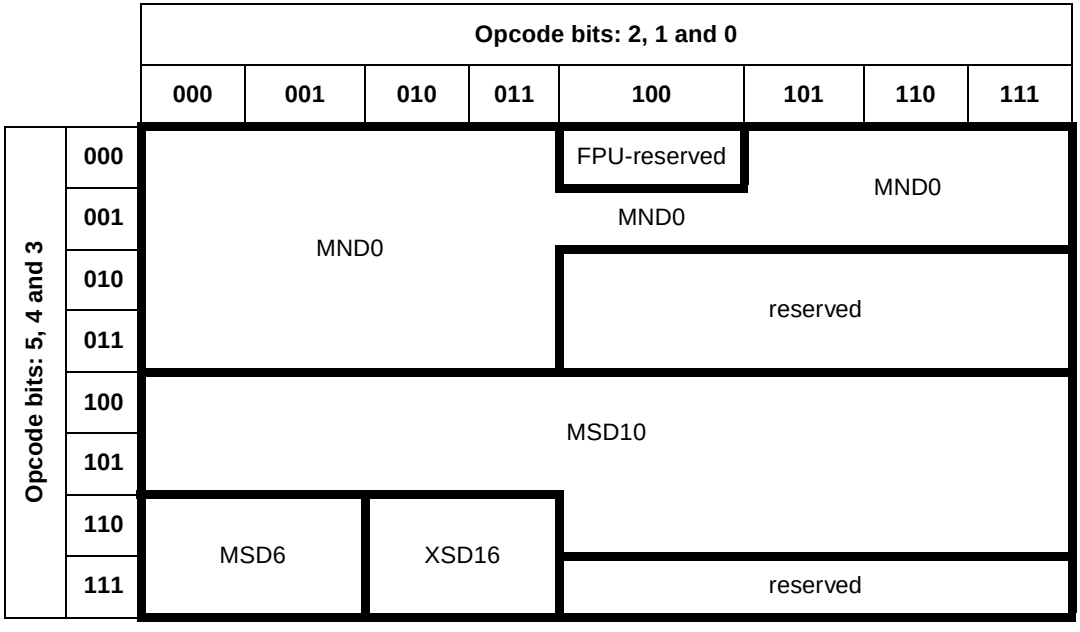


Table 36: Opcodes and major formats

A.2 Opcode assignment

The opcode allocation is shown in *Table 37*.

Opcode bits: 5, 4 and 3	Opcode bits: 2, 1 and 0							
	000	**001**	**010**	**011**	**100**	**101**	**110**	**111**
000	ALU	ALU	MM	MM	FPU-reserved	FPU	FPU	FLOAD
001	RMW	MISC	MM	MM	FPU	FPU	FPU	FSTORE
010	LOAD	BRANCH	MM	MM	reserved	reserved	reserved	reserved
011	STORE	BRANCH	PT	MISC	reserved	reserved	reserved	reserved
100	LD.B	LD.W	LD.L	LD.Q	LD.UB	FLD.S	FLD.P	FLD.D
101	ST.B	ST.W	ST.L	ST.Q	LD.UW	FST.S	FST.P	FST.D
110	LOAD	ALU	SHORI	MOVI	ADDI	ADDI.L	ANDI	ORI
111	STORE	BRANCH	PTA	PTB	reserved	reserved	reserved	reserved

Table 37: Opcode allocation

For major formats MSD10 and XSD16, this table contains the instruction names allocated to each opcode. For major formats MND0 and MSD6, this table gives an italicized name which identified the group of the instructions assigned to that opcode. These opcodes have 4 extension bits which can distinguish up to 16 instructions. The assignment of these extension bits is described in the following sections. This table uses the following abbreviations for groups of instructions:

1 ALU: straightforward arithmetic or bitwise instructions.

2 MM: multimedia instructions.

3 FPU: floating-point instructions (not load or store).

4 FLOAD: floating-point load instructions,

5 FSTORE: floating-point store instructions.

6 RMW: instructions that read-modify-write a register operand.

7 MISC: miscellaneous instructions.

8 LOAD: general-purpose load instructions.

9 STORE: general-purpose store instructions.

10 BRANCH: instructions that read target registers.

11 PT: instructions that write target registers.

A.3 Reserved bits [0, 3]

Bits [0,3] of every instruction are reserved for the future expansion of the instruction set architecture. They must be set to 0b0000 on all instructions. In the current architecture specification, execution of an instruction with a non-zero value in bits [0,3] leads to a reserved instruction exception. This exception check is performed prior to decoding the opcode and extension opcode of the instruction.

Software should not rely on the reserved instruction exception generated for incorrect settings of bits [0, 3]. On a future implementation, the behavior for non-zero values could be modified to add new mechanism to the architecture.

A.4 Reserved instructions

There are 13 opcodes marked as 'reserved' or 'FPU-reserved' in *Table 37: Opcode allocation on page 306*. These contain no instructions and are reserved for future expansion of the instruction set.

Major formats MND0 and MSD6 use an additional 4-bit extension field to distinguish up to 16 instructions per opcode. The combination of the 6-bit opcode and the 4-bit extension opcode is called a 10-bit extended opcode.

There are many extended opcode values which are also reserved for future expansion of the instruction set. These cases are indicated by empty cells in the opcode assignment tables in *Appendix A: Major format MND0 on page 310* and *Appendix A: Major format MSD6 on page 322*.

Execution of a reserved opcode leads to a reserved instruction exception:

- The SHmedia instruction with encoding 0x6FF4FFF0 is guaranteed to be reserved on all implementations. Execution of this SHmedia instruction will always result in a RESINST exception. This instruction corresponds to an opcode of 011011 and an extension opcode of 0100.

- Software should not rely on a RESINST exception for the execution of other reserved opcodes. On a future implementation, any of these reserved opcodes can be used to expand the instruction set.

A.5 Reserved operand bits

Some SHmedia instructions have operand fields where some or all of the bits are unused. These are reserved operand bits and must be set to appropriate values (as defined in *Volume 1 Chapter 4: SHmedia instructions*), otherwise the behavior is architecturally undefined.

A.6 Floating-point instructions

A reserved instruction exception is raised when there is a non-zero value in bits [0, 3] of an instruction, as described in *Appendix A: Reserved bits [0, 3] on page 307*. When these bits are all zero, the instruction set is partitioned into general-purpose instructions and floating-point instructions. The floating-point instruction set consists of the 14 opcodes from *Table 37* that start with the letter 'F'. The general-purpose instruction set consists of the remaining opcodes.

An implementation can choose not to provide floating-point and SR.FD will then always read as 1. If an implementation provides floating-point, software can disable it by setting the SR.FD flag. In both of these cases, execution of an instruction from the floating-point instruction set leads to an FPU disabled exception.

Note that if an instruction has a non-zero value in bits [0, 3] and also has an opcode from *Table 37* that starts with the letter 'F', then it is not considered to be an FPU instruction. Execution of this instruction will result in a reserved instruction exception, not an FPU disabled exception, regardless of whether the floating-point unit is present or disabled.

In other cases, the FPU disabled exception takes precedence over the reserved instruction exception. Thus, execution of a reserved floating-point instruction where the floating-point instruction set is not available, leads to a floating-point disabled exception.

A.7 Minor formats

The minor formats are packed into opcodes as shown in *Table 38*.

Opcode bits: 5, 4 and 3	Opcode bits: 2, 1 and 0							
	000	**001**	**010**	**011**	**100**	**101**	**110**	**111**
000	mnd0 mxd0	mnd0	mnd0	mnd0	FPU-reserved	ghf0	gxf0	mxf0 xxf0 mnf0
001	mnw0	kxd0	mnd0 mxd0	mnd0	gxd0 ghd0 gxx0	ghf0 ghq0	gxf0	mnz0
010	mnd0	bxd0	mnw0	mnd0	reserved	reserved	reserved	reserved
011	mny0	mnc0	xna0	mxj0 mxx0 xxx0	reserved	reserved	reserved	reserved
100	msd10	msd10	msd10	msd10	msd10	msf10	msf10	msf10
101	msy10	msy10	msy10	msy10	msy10	msz10	msz10	msz10
110	msd6	msd6	xsw16	xsd16	msd10	msd10	msd10	msd10
111	msx6 msy6	msc6	xsa16	xsa16	reserved	reserved	reserved	reserved

Table 38: Opcodes and minor formats

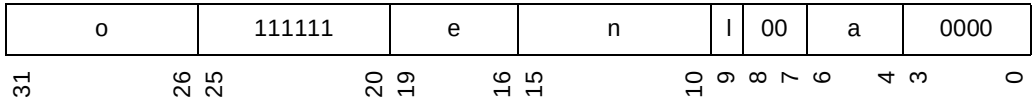
A.8 Major format MND0

This major format contains the minor formats listed in the following table.

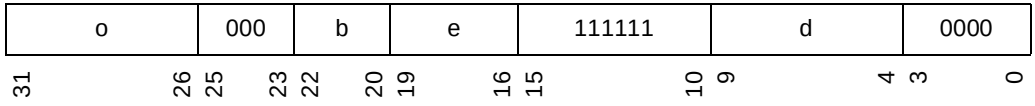
Format Name	Example Mnemonic(s)		Operands	
xna0	PTABS		R_n, T_a	
bxd0	BLINK		T_b, R_d	
kxd0	GETCON		C_k, R_d	
mnd0	ADD		R_m, R_n, R_d	
mxd0	BYTEREV		R_m, R_d	
mxj0	PUTCON		R_m, C_j	
xxx0	RTE			
mnw0	CMVEQ		R_m, R_n, R_w	
mnc0	BEQ		R_m, R_n, T_c	
mny0	STX.B		R_m, R_n, R_y	
mxx0	TRAPA		R_m	
xxf0	FGETSCR		F_f	
gxx0	FPUTSCR		F_g	
ghq0	FMAC.S		F_g, F_h, F_q	
ghf0	FADD.S	FADD.D	F_g, F_h, F_f	D_g, D_h, D_f
ghd0	FCMPEQ.S	FCMPEQ.D	F_g, F_h, R_d	D_g, D_h, R_d
gxf0	FSQRT.S	FSQRT.D	F_g, F_f	D_g, D_f
gxd0	FMOV.SL	FMOV.DQ	F_g, R_d	D_g, R_d
mxf0	FMOV.LS	FMOV.QD	R_m, F_f	R_m, D_f
mnf0	FLDX.S	FLDX.D	R_m, R_n, F_f	R_m, R_n, D_f
mnz0	FSTX.S	FSTX.D	R_m, R_n, F_z	R_m, R_n, D_z

The bit allocation of these minor formats is described by the subsequent diagrams.

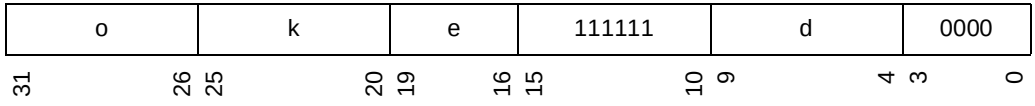
xna0



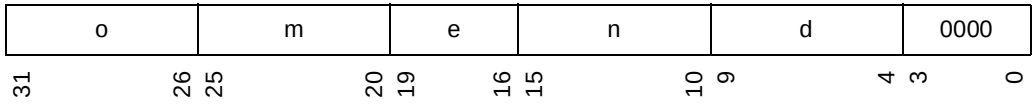
bxd0



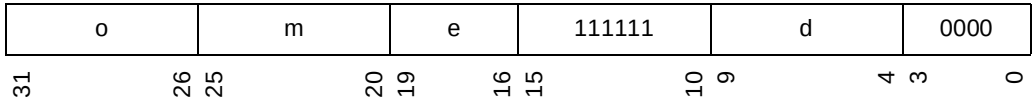
kxd0



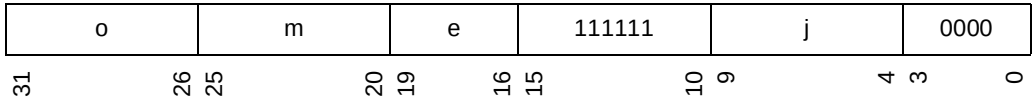
mnd0



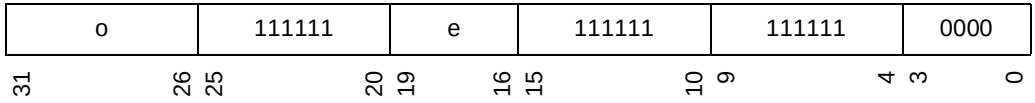
mxd0



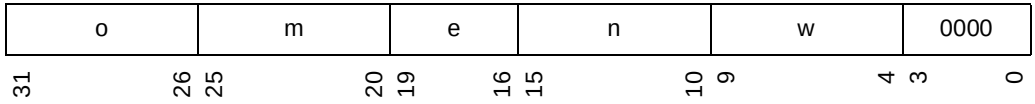
mxj0



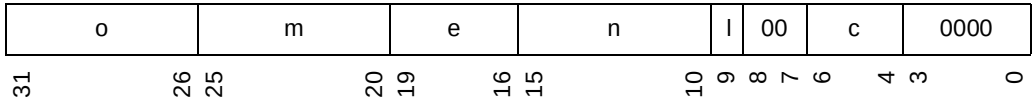
xxx0



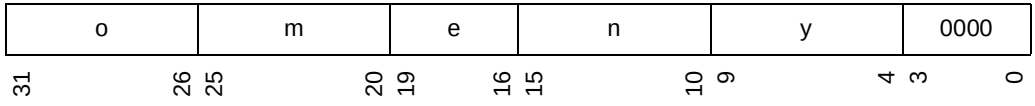
mnw0



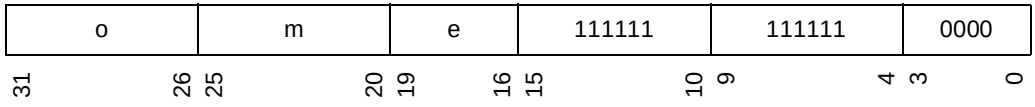
mnc0



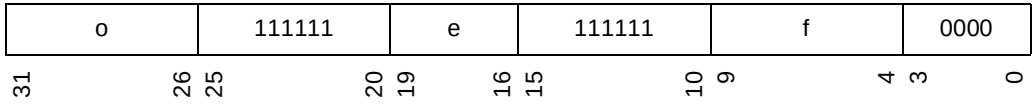
mny0



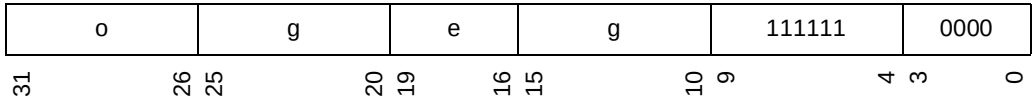
mxx0



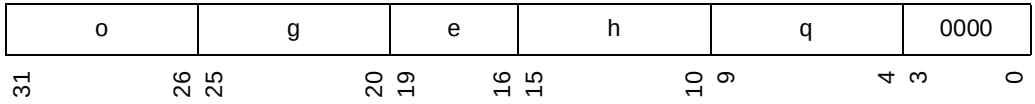
xxf0



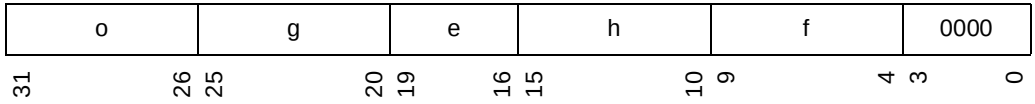
gxx0



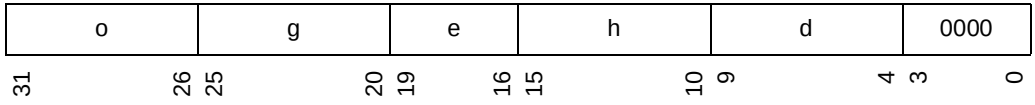
ghq0



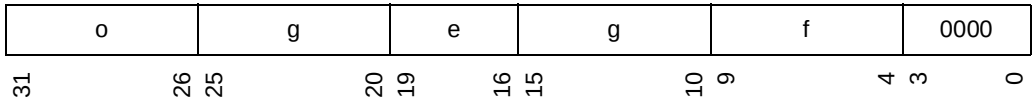
ghf0



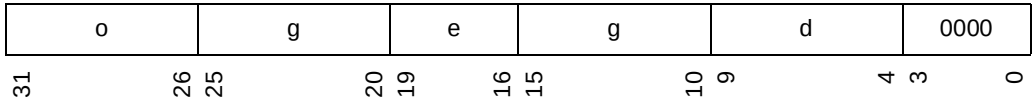
ghd0



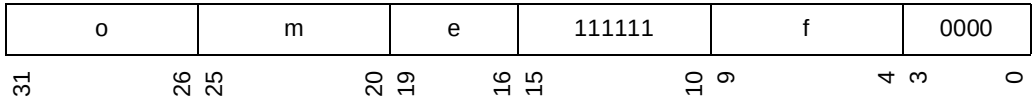
gxf0



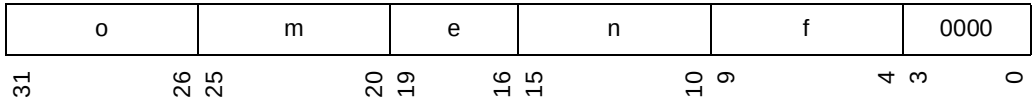
gxd0



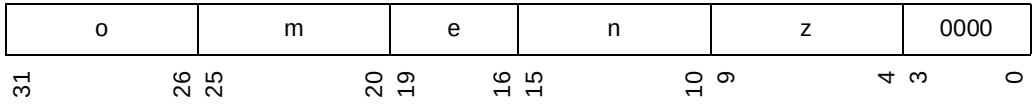
mxf0



mnf0



mnz0



The MSD0 format contains 4 extension bits. Each opcode can support up to 16 instructions. These are assigned by the following tables.

		Extension opcode bits: 1 and 0			
		00	**01**	**10**	**11**
Bits: 3 and 2	**00**		CMPEQ		CMPGT
	01				CMPGTU
	10	ADD.L	ADD	SUB.L	SUB
	11	ADDZ.L	NSB	MULU.L	BYTEREV

Table 39: Opcode 000000, ALU

		Extension opcode bits: 1 and 0			
		00	**01**	**10**	**11**
Bits: 3 and 2	**00**	SHLLD.L	SHLLD	SHLRD.L	SHLRD
	01			SHARD.L	SHARD
	10		OR		AND
	11		XOR	MULS.L	ANDC

Table 40: Opcode 000001, ALU

		Extension opcode bits: 1 and 0			
		00	**01**	**10**	**11**
Bits: 3 and 2	**00**		MADD.W	MADD.L	
	01	MADDS.UB	MADDS.W	MADDS.L	
	10		MSUB.W	MSUB.L	
	11	MSUBS.UB	MSUBS.W	MSUBS.L	

Table 41: Opcode 000010, MM

		Extension opcode bits: 1 and 0			
		00	**01**	**10**	**11**
Bits: 3 and 2	**00**		MSHLLD.W	MSHLLD.L	
	01		MSHALDS.W	MSHALDS.L	
	10		MSHARD.W	MSHARD.L	MSHARDS.Q
	11		MSHLRD.W	MSHLRD.L	

Table 42: Opcode 000011, MM

		Extension opcode bits: 1 and 0			
		00	01	10	11
Bits: 3 and 2	00				
	01			FIPR.S	
	10				
	11			FTRV.S	

Table 43: Opcode 000101, FPU

		Extension opcode bits: 1 and 0			
		00	01	10	11
Bits: 3 and 2	00	FABS.S	FABS.D	FNEG.S	FNEG.D
	01				
	10	FSINA.S		FSRRA.S	
	11	FCOSA.S			

Table 44: Opcode 000110, FPU

		Extension opcode bits: 1 and 0			
		00	01	10	11
Bits: 3 and 2	00	FMOV.LS	FMOV.QD	FGETSCR	
	01				
	10	FLDX.S	FLDX.D		
	11		FLDX.P		

Table 45: Opcode 000111, FLOAD

		Extension opcode bits: 1 and 0			
		00	01	10	11
Bits: 3 and 2	00		CMVEQ		SWAP.Q
	01		CMVNE		
	10				
	11				

Table 46: Opcode 001000, RMW

		Extension opcode bits: 1 and 0			
		00	01	10	11
Bits: 3 and 2	00				
	01				
	10				
	11				GETCON

Table 47: Opcode 001001, MISC

		Extension opcode bits: 1 and 0			
		00	01	10	11
Bits: 3 and 2	00	MCMPEQ.B	MCMPEQ.W	MCMPEQ.L	
	01	MCMPGT.UB	MCMPGT.W	MCMPGT.L	MEXTR1
	10		MABS.W	MABS.L	MEXTR2
	11		MPERM.W		MEXTR3

Table 48: Opcode 001010, MM

		Extension opcode bits: 1 and 0			
		00	01	10	11
Bits: 3 and 2	00	MSHFLO.B	MSHFLO.W	MSHFLO.L	MEXTR4
	01	MSHFHI.B	MSHFHI.W	MSHFHI.L	MEXTR5
	10				MEXTR6
	11				MEXTR7

Table 49: Opcode 001011, MM

		Extension opcode bits: 1 and 0			
		00	01	10	11
Bits: 3 and 2	00	FMOV.SL	FMOV.DQ	FPUTSCR	
	01				
	10	FCMPEQ.S	FCMPEQ.D	FCMPUN.S	FCMPUN.D
	11	FCMPGT.S	FCMPGT.D	FCMPGE.S	FCMPGE.D

Table 50: Opcode 001100, FPU

		Extension opcode bits: 1 and 0			
		00	01	10	11
Bits: 3 and 2	00	FADD.S	FADD.D	FSUB.S	FSUB.D
	01	FDIV.S	FDIV.D	FMUL.S	FMUL.D
	10				
	11			FMAC.S	

Table 51: Opcode 001101, FPU

		Extension opcode bits: 1 and 0			
		00	01	10	11
Bits: 3 and 2	00	FMOV.S	FMOV.D		
	01	FSQRT.S	FSQRT.D	FCNV.SD	FCNV.DS
	10	FTRC.SL	FTRC.DQ	FTRC.SQ	FTRC.DL
	11	FLOAT.LS	FLOAT.QD	FLOAT.LD	FLOAT.QS

Table 52: Opcode 001110, FPU

		Extension opcode bits: 1 and 0			
		00	01	10	11
Bits: 3 and 2	00				
	01				
	10	FSTX.S	FSTX.D		
	11		FSTX.P		

Table 53: Opcode 001111, FSTORE

		Extension opcode bits: 1 and 0			
		00	01	10	11
Bits: 3 and 2	00	LDX.B	LDX.W	LDX.L	LDX.Q
	01	LDX.UB	LDX.UW		
	10				
	11				

Table 54: Opcode 010000, LOAD

		Extension opcode bits: 1 and 0			
		00	01	10	11
Bits: 3 and 2	00	BLINK			
	01	GETTR			
	10				
	11				

Table 55: Opcode 010001, BRANCH

		Extension opcode bits: 1 and 0			
		00	01	10	11
Bits: 3 and 2	00	MSAD.UBQ	MMACFX.WL		MCMV
	01		MMACNFX.WL		
	10		MMULSUM.WQ		
	11				

Table 56: Opcode 010010, MM

		Extension opcode bits: 1 and 0			
		00	01	10	11
Bits: 3 and 2	00		MMUL.W	MMUL.L	
	01		MMULFX.W	MMULFX.L	
	10	MCNVS.WB	MMULFXRP.W	MMULLO.WL	
	11	MCNVS.WUB	MCNVS.LW	MMULHI.WL	

Table 57: Opcode 010011, MM

		Extension opcode bits: 1 and 0			
		00	01	10	11
Bits: 3 and 2	00	STX.B	STX.W	STX.L	STX.Q
	01				
	10				
	11				

Table 58: Opcode 011000, STORE

		Extension opcode bits: 1 and 0			
		00	01	10	11
Bits: 3 and 2	00		BEQ		BGE
	01		BNE		BGT
	10				BGEU
	11				BGTU

Table 59: Opcode 011001, BRANCH

		Extension opcode bits: 1 and 0			
		00	01	10	11
Bits: 3 and 2	00		PTABS		
	01		PTREL		
	10				
	11				

Table 60: Opcode 011010, PT

		Extension opcode bits: 1 and 0			
		00	**01**	**10**	**11**
Bits: 3 and 2	**00**	NOP	TRAPA	SYNCI	RTE
	01		BRK	SYNCO	SLEEP
	10				
	11				PUTCON

Table 61: Opcode 011011, MISC

A.9 Major format MSD6

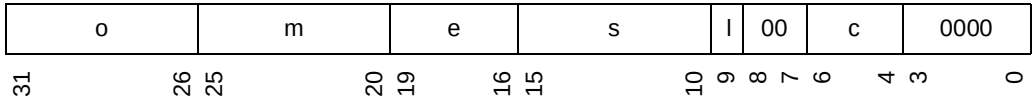
This major format contains the minor formats given in *Table 62*.

Format Name	Example Mnemonic	Operands
msc6	BEQI	R_m, s_6, T_c
msd6	SHLLI	R_m, s_6, R_d
msx6	ALLOCO	R_m, s_6
msy6	STHI.L	R_m, s_6, R_y

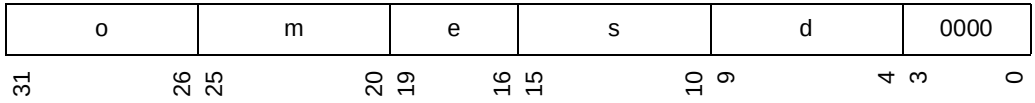
Table 62: MSD6 format summary

The bit allocation of these minor formats is described by the subsequent diagrams.

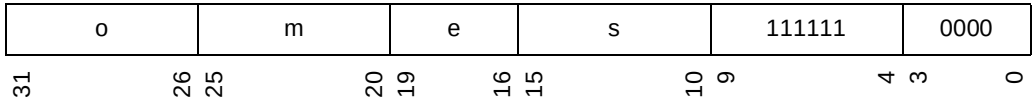
msc6



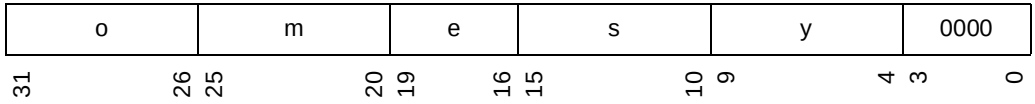
msd6



msx6



msy6



The MSD6 format contains 4 extension bits. Each opcode can support up to 16 instructions. These are assigned by the following tables.

		Extension opcode bits: 1 and 0			
		00	**01**	**10**	**11**
Bits: 3 and 2	**00**			LDLO.L	LDLO.Q
	01			LDHI.L	LDHI.Q
	10				
	11				GETCFG

Table 63: Opcode 110000, LOAD

		Extension opcode bits: 1 and 0			
		00	01	10	11
Bits: 3 and 2	00	SHLLI.L	SHLLI	SHLRI.L	SHLRI
	01			SHARI.L	SHARI
	10				
	11		XORI		

Table 64: Opcode 110001, ALU

		Extension opcode bits: 1 and 0			
		00	01	10	11
Bits: 3 and 2	00		PREFI	STLO.L	STLO.Q
	01	ALLOCO	ICBI	STHI.L	STHI.Q
	10	OCBP	OCBI		
	11	OCBWB			PUTCFG

Table 65: Opcode 111000, STORE

		Extension opcode bits: 1 and 0			
		00	01	10	11
Bits: 3 and 2	00		BEQI		
	01		BNEI		
	10				
	11				

Table 66: Opcode 111001, BRANCH

A.10 Major format MSD10

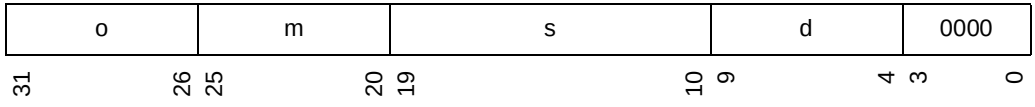
This major format contains the minor formats given in *Table 67*

Format Name	Example Mnemonic(s)		Operands	
msd10	LD.B		R_m, s_{10}, R_d	
msy10	ST.B		R_m, s_{10}, R_y	
msf10	FLD.S	FLD.D	R_m, s_{10}, F_f	R_m, s_{10}, D_f
msz10	FST.S	FST.D	R_m, s_{10}, F_z	R_m, s_{10}, D_z

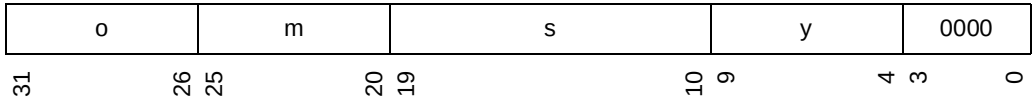
Table 67: MSD10 format summary

These formats contain no extension bits. There is 1 instruction for each opcode.

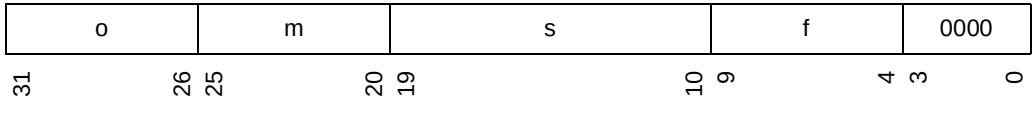
msd10



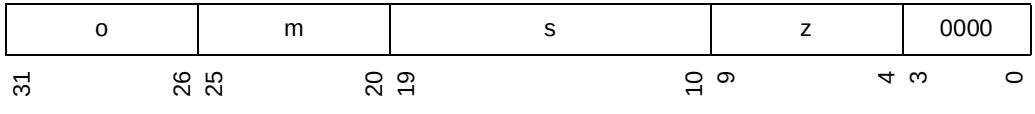
msy10



msf10



msz10



A.11 Major format XSD16

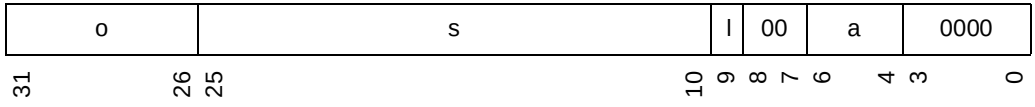
This major format contains the minor formats given in *Table 68*.

Format Name	Example Mnemonic	Operands
xsa16	PTA	s_{16}, T_a
xsd16	MOVI	s_{16}, R_d
xsw16	SHORI	s_{16}, R_w

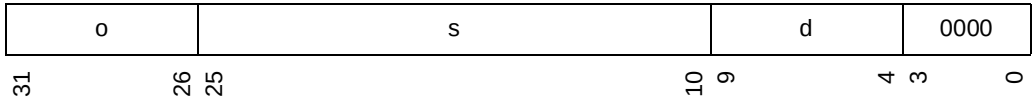
Table 68: XSD16 format summary

These formats contain no extension bits. There is 1 instruction for each opcode.

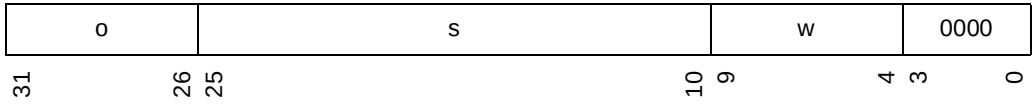
xsa16



xsd16



xsw16



SuperH

Index

G

I

L

M

N

O

P

R

S

T

U

W

XYZ