# MIPS® Architecture for Programmers Volume IV-j: The MIPS64® SIMD Architecture Module

**Document Number: MD00868**
**Revision 1.12**
**February 3, 2016**

Template: nB1.02, Built with tags: ARCH FPU_PS FPU_PSandARCH

# Contents

# List of Figures

# List of Tables

*Chapter 1*

# About This Book

The MIPS® Architecture for Programmers Volume IV-j: The MIPS64® SIMD Architecture Module comes as part of a multi-volume set.

- Volume I-A describes conventions used throughout the document set, and provides an introduction to the MIPS64® Architecture
- Volume I-B describes conventions used throughout the document set, and provides an introduction to the microMIPS64™ Architecture
- Volume II-A provides detailed descriptions of each instruction in the MIPS64® instruction set
- Volume II-B provides detailed descriptions of each instruction in the microMIPS64™ instruction set
- Volume III describes the MIPS64® and microMIPS64™ Privileged Resource Architecture which defines and governs the behavior of the privileged resources included in a MIPS® processor implementation
- Volume IV-a describes the MIPS16e™ Application-Specific Extension to the MIPS64® Architecture. Beginning with Release 3 of the Architecture, microMIPS is the preferred solution for smaller code size.
- Volume IV-b describes the MDMX™ Application-Specific Extension to the MIPS64® Architecture and microMIPS64™. With Release 5 of the Architecture, MDMX is deprecated. MDMX and MSA can not be implemented at the same time.
- Volume IV-c describes the MIPS-3D® Application-Specific Extension to the MIPS® Architecture
- Volume IV-d describes the SmartMIPS®Application-Specific Extension to the MIPS32® Architecture and the microMIPS32™ Architecture and is not applicable to the MIPS64® document set nor the microMIPS64™ document set.
- Volume IV-e describes the MIPS® DSP Module to the MIPS® Architecture
- Volume IV-f describes the MIPS® MT Module to the MIPS® Architecture
- Volume IV-h describes the MIPS® MCU Application-Specific Extension to the MIPS® Architecture
- Volume IV-i describes the MIPS® Virtualization Module to the MIPS® Architecture
- Volume IV-j describes the MIPS® SIMD Architecture Module to the MIPS® Architecture

## 1.1 Typographical Conventions

This section describes the use of *italic*, **bold** and `courier` fonts in this book.

### 1.1.1 Italic Text

- is used for *emphasis*
- is used for *bits*, *fields*, *registers*, that are important from a software perspective (for instance, address bits used by software, and programmable fields and registers), and various *floating point instruction formats*, such as *S*, *D*, and *PS*
- is used for the memory access types, such as *cached* and *uncached*

### 1.1.2 Bold Text

- represents a term that is being **defined**
- is used for **bits** and **fields** that are important from a hardware perspective (for instance, **register** bits, which are not programmable but accessible only to hardware)
- is used for ranges of numbers; the range is indicated by an ellipsis. For instance, **5..1** indicates numbers 5 through 1
- is used to emphasize **UNPREDICTABLE** and **UNDEFINED** behavior, as defined below.

### 1.1.3 Courier Text

`Courier` fixed-width font is used for text that is displayed on the screen, and for examples of code and instruction pseudocode.

## 1.2 UNPREDICTABLE and UNDEFINED

The terms **UNPREDICTABLE** and **UNDEFINED** are used throughout this book to describe the behavior of the processor in certain cases. **UNDEFINED** behavior or operations can occur only as the result of executing instructions in a privileged mode (i.e., in Kernel Mode or Debug Mode, or with the CP0 usable bit set in the Status register). Unprivileged software can never cause **UNDEFINED** behavior or operations. Conversely, both privileged and unprivileged software can cause **UNPREDICTABLE** results or operations.

### 1.2.1 UNPREDICTABLE

**UNPREDICTABLE** results may vary from processor implementation to implementation, instruction to instruction, or as a function of time on the same implementation or instruction. Software can never depend on results that are **UNPREDICTABLE**. **UNPREDICTABLE** operations may cause a result to be generated or not. If a result is generated, it is **UNPREDICTABLE**. **UNPREDICTABLE** operations may cause arbitrary exceptions.

**UNPREDICTABLE** results or operations have several implementation restrictions:

- Implementations of operations generating **UNPREDICTABLE** results must not depend on any data source (memory or internal state) which is inaccessible in the current processor mode
- **UNPREDICTABLE** operations must not read, write, or modify the contents of memory or internal state which is inaccessible in the current processor mode. For example, **UNPREDICTABLE** operations executed in user mode must not access memory or internal state that is only accessible in Kernel Mode or Debug Mode or in another process

- **UNPREDICTABLE** operations must not halt or hang the processor

### 1.2.2 UNDEFINED

**UNDEFINED** operations or behavior may vary from processor implementation to implementation, instruction to instruction, or as a function of time on the same implementation or instruction. **UNDEFINED** operations or behavior may vary from nothing to creating an environment in which execution can no longer continue. **UNDEFINED** operations or behavior may cause data loss.

**UNDEFINED** operations or behavior has one implementation restriction:

- **UNDEFINED** operations or behavior must not cause the processor to hang (that is, enter a state from which there is no exit other than powering down the processor). The assertion of any of the reset signals must restore the processor to an operational state

### 1.2.3 UNSTABLE

**UNSTABLE** results or values may vary as a function of time on the same implementation or instruction. Unlike **UNPREDICTABLE** values, software may depend on the fact that a sampling of an **UNSTABLE** value results in a legal transient value that was correct at some point in time prior to the sampling.

**UNSTABLE** values have one implementation restriction:

- Implementations of operations generating **UNSTABLE** results must not depend on any data source (memory or internal state) which is inaccessible in the current processor mode

## 1.3 Special Symbols in Pseudocode Notation

In this book, algorithmic descriptions of an operation are described as pseudocode in a high-level language notation resembling Pascal. Special symbols used in the pseudocode notation are listed in Table 1.1.

**Table 1.1 Symbols Used in Instruction Operation Statements**

| Symbol | Meaning |
|---|---|
| ← | Assignment |
| =, ≠ | Tests for equality and inequality |
| \|\| | Bit string concatenation |
| $x^y$ | A *y*-bit string formed by *y* copies of the single-bit value *x* |
| b#n | A constant value *n* in base *b*. For instance 10#100 represents the decimal value 100, 2#100 represents the binary value 100 (decimal 4), and 16#100 represents the hexadecimal value 100 (decimal 256). If the "b#" prefix is omitted, the default base is 10. |
| 0bn | A constant value *n* in base *2*. For instance 0b100 represents the binary value 100 (decimal 4). |
| 0xn | A constant value *n* in base *16*. For instance 0x100 represents the hexadecimal value 100 (decimal 256). |
| $x_{y\ z}$ | Selection of bits *y* through *z* of bit string *x*. Little-endian bit notation (rightmost bit is 0) is used. If *y* is less than *z*, this expression is an empty (zero length) bit string. |
| +, – | 2's complement or floating point arithmetic: addition, subtraction |

**Table 1.1 Symbols Used in Instruction Operation Statements (Continued)**

| Symbol | Meaning |
|---|---|
| *, ∞ | 2's complement or floating point multiplication (both used for either) |
| div | 2's complement integer division |
| mod | 2's complement modulo |
| / | Floating point division |
| < | 2's complement less-than comparison |
| > | 2's complement greater-than comparison |
| ≤ | 2's complement less-than or equal comparison |
| ≥ | 2's complement greater-than or equal comparison |
| nor | Bitwise logical NOR |
| xor | Bitwise logical XOR |
| and | Bitwise logical AND |
| or | Bitwise logical OR |
| not | Bitwise inversion |
| && | Logical (non-Bitwise) AND |
| << | Logical Shift left (shift in zeros at right-hand-side) |
| >> | Logical Shift right (shift in zeros at left-hand-side) |
| GPRLEN | The length in bits (32 or 64) of the CPU general-purpose registers |
| *GPR[x]* | CPU general-purpose register *x*. The content of *GPR[0]* is always zero. In Release 2 of the Architecture, GPR[x] is a short-hand notation for $SGPR[ SRSCtl_{CSS}, x]$. |
| SGPR[s,x] | In Release 2 of the Architecture and subsequent releases, multiple copies of the CPU general-purpose registers may be implemented. *SGPR[s,x]* refers to GPR set *s*, register *x*. |
| *FPR[x]* | Floating Point operand register *x* |
| *FCC[CC]* | Floating Point condition code CC. *FCC[0]* has the same value as *COC[1]*. |
| *FPR[x]* | Floating Point (Coprocessor unit 1), general register *x* |
| *CPR[z,x,s]* | Coprocessor unit *z*, general register *x*, select *s* |
| CP2CPR[x] | Coprocessor unit 2, general register *x* |
| *CCR[z,x]* | Coprocessor unit *z*, control register *x* |
| CP2CCR[x] | Coprocessor unit 2, control register *x* |
| *COC[z]* | Coprocessor unit *z* condition signal |
| *Xlat[x]* | Translation of the MIPS16e GPR number *x* into the corresponding 32-bit GPR number |
| BigEndianMem | Endian mode as configured at chip reset (0 →Little-Endian, 1 → Big-Endian). Specifies the endianness of the memory interface (see LoadMemory and StoreMemory pseudocode function descriptions), and the endianness of Kernel and Supervisor mode execution. |
| BigEndianCPU | The endianness for load and store instructions (0 → Little-Endian, 1 → Big-Endian). In User mode, this endianness may be switched by setting the *RE* bit in the *Status* register. Thus, BigEndianCPU may be computed as (BigEndianMem XOR ReverseEndian). |
| ReverseEndian | Signal to reverse the endianness of load and store instructions. This feature is available in User mode only, and is implemented by setting the *RE* bit of the *Status* register. Thus, ReverseEndian may be computed as ($SR_{RE}$ and User mode). |

**Table 1.1 Symbols Used in Instruction Operation Statements (Continued)**

| Symbol | Meaning |
|---|---|
| *LLbit* | Bit of **virtual** state used to specify operation for instructions that provide atomic read-modify-write. *LLbit* is set when a linked load occurs and is tested by the conditional store. It is cleared, during other CPU operation, when a store to the location would no longer be atomic. In particular, it is cleared by exception return instructions. |
| **I**:,<br>**I+n**:,<br>**I-n**: | This occurs as a prefix to *Operation* description lines and functions as a label. It indicates the instruction time during which the pseudocode appears to "execute." Unless otherwise indicated, all effects of the current instruction appear to occur during the instruction time of the current instruction. No label is equivalent to a time label of **I**. Sometimes effects of an instruction appear to occur either earlier or later — that is, during the instruction time of another instruction. When this happens, the instruction operation is written in sections labeled with the instruction time, relative to the current instruction **I**, in which the effect of that pseudocode appears to occur. For example, an instruction may have a result that is not available until after the next instruction. Such an instruction has the portion of the instruction operation description that writes the result register in a section labeled **I+1**.<br>The effect of pseudocode statements for the current instruction labelled **I+1** appears to occur "at the same time" as the effect of pseudocode statements labeled **I** for the following instruction. Within one pseudocode sequence, the effects of the statements take place in order. However, between sequences of statements for different instructions that occur "at the same time," there is no defined order. Programs must not depend on a particular order of evaluation between such sections. |
| PC | The *Program Counter* value. During the instruction time of an instruction, this is the address of the instruction word. The address of the instruction that occurs during the next instruction time is determined by assigning a value to *PC* during an instruction time. If no value is assigned to *PC* during an instruction time by any pseudocode statement, it is automatically incremented by either 2 (in the case of a 16-bit MIPS16e instruction) or 4 before the next instruction time. A taken branch assigns the target address to the *PC* during the instruction time of the instruction in the branch delay slot.<br>In the MIPS Architecture, the PC value is only visible indirectly, such as when the processor stores the restart address into a GPR on a jump-and-link or branch-and-link instruction, or into a Coprocessor 0 register on an exception. The PC value contains a full 64-bit address all of which are significant during a memory reference. |
| ISA Mode | In processors that implement the MIPS16e Application Specific Extension or the microMIPS base architectures, the *ISA Mode* is a single-bit register that determines in which mode the processor is executing, as follows:<br><br>**Encoding** / **Meaning**<br>0 — The processor is executing 32-bit MIPS instructions<br>1 — The processor is executing MIIPS16e or microMIPS instructions<br><br>In the MIPS Architecture, the ISA Mode value is only visible indirectly, such as when the processor stores a combined value of the upper bits of PC and the ISA Mode into a GPR on a jump-and-link or branch-and-link instruction, or into a Coprocessor 0 register on an exception. |
| PABITS | The number of physical address bits implemented is represented by the symbol PABITS. As such, if 36 physical address bits were implemented, the size of the physical address space would be $2^{PABITS} = 2^{36}$ bytes. |
| SEGBITS | The number of virtual address bits implemented in a segment of the address space is represented by the symbol SEGBITS. As such, if 40 virtual address bits are implemented in a segment, the size of the segment is $2^{SEGBITS} = 2^{40}$ bytes. |

**Table 1.1 Symbols Used in Instruction Operation Statements (Continued)**

| Symbol | Meaning |
|---|---|
| FP32RegistersMode | Indicates whether the FPU has 32-bit or 64-bit floating point registers (FPRs). In MIPS32 Release 1, the FPU has 32 32-bit FPRs in which 64-bit data types are stored in even-odd pairs of FPRs. In MIPS64, (and optionally in MIPS32 Release2 and MIPSr3) the FPU has 32 64-bit FPRs in which 64-bit data types are stored in any FPR.<br><br>In MIPS32 Release 1 implementations, **FP32RegistersMode** is always a 0. MIPS64 implementations have a compatibility mode in which the processor references the FPRs as if it were a MIPS32 implementation. In such a case **FP32RegisterMode** is computed from the FR bit in the *Status* register. If this bit is a 0, the processor operates as if it had 32 32-bit FPRs. If this bit is a 1, the processor operates with 32 64-bit FPRs. The value of **FP32RegistersMode** is computed from the FR bit in the *Status* register. |
| InstructionInBranchDelaySlot | Indicates whether the instruction at the Program Counter address was executed in the delay slot of a branch or jump. This condition reflects the *dynamic* state of the instruction, not the *static* state. That is, the value is false if a branch or jump occurs to an instruction whose PC immediately follows a branch or jump, but which is not executed in the delay slot of a branch or jump. |
| SignalException(exception, argument) | Causes an exception to be signaled, using the exception parameter as the type of exception and the argument parameter as an exception-specific argument). Control does not return from this pseudocode function—the exception is signaled at the point of the call. |

## 1.4 For More Information

Various MIPS RISC processor manuals and additional information about MIPS products can be found at the MIPS URL: http://www mips.com

For comments or questions on the MIPS64® Architecture or this document, send Email to support@mips.com.

*Chapter 2*

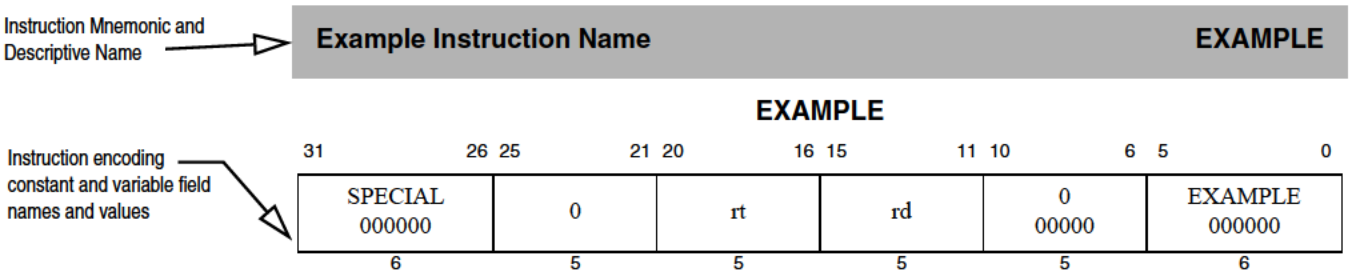
# Guide to the Instruction Set

This chapter provides a detailed guide to understanding the instruction descriptions, which are listed in alphabetical order in the tables at the beginning of the next chapter.

## 2.1 Understanding the Instruction Fields

Figure 2.1 shows an example instruction. Following the figure are descriptions of the fields listed below:

**Figure 2.1 Example of Instruction Description**

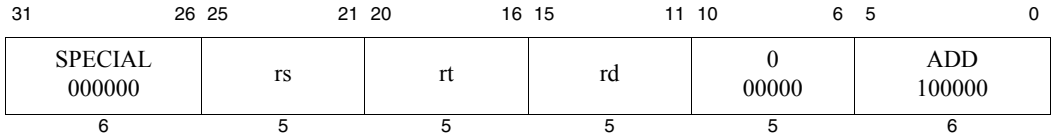



## 2.1.1 Instruction Fields

Fields encoding the instruction word are shown in register form at the top of the instruction description. The following rules are followed:

- The values of constant fields and the *opcode* names are listed in uppercase (SPECIAL and ADD in Figure 2.2). Constant values in a field are shown in binary below the symbolic or hexadecimal value.
- All variable fields are listed with the lowercase names used in the instruction description (*rs*, *rt*, and *rd* in Figure 2.2).
- Fields that contain zeros but are not named are unused fields that are required to be zero (bits 10:6 in Figure 2.2). If such fields are set to non-zero values, the operation of the processor is **UNPREDICTABLE**.

**Figure 2.2 Example of Instruction Fields**




## 2.1.2 Instruction Descriptive Name and Mnemonic

The instruction descriptive name and mnemonic are printed as page headings for each instruction, as shown in Figure 2.3.

**Figure 2.3 Example of Instruction Descriptive Name and Mnemonic**


## 2.1.3 Format Field

The assembler formats for the instruction and the architecture level at which the instruction was originally defined are given in the *Format* field. If the instruction definition was later extended, the architecture levels at which it was extended and the assembler formats for the extended definition are shown in their order of extension (for an example, see C.cond fmt). The MIPS architecture levels are inclusive; higher architecture levels include all instructions in previous levels. Extensions to instructions are backwards compatible. The original assembler formats are valid for the extended architecture.

**Figure 2.4 Example of Instruction Format**

**Format:** `ADD fd,rs,rt` **MIPS32**

The assembler format is shown with literal parts of the assembler instruction printed in uppercase characters. The variable parts, the operands, are shown as the lowercase names of the appropriate fields. The architectural level at which the instruction was first defined, for example “MIPS32” is shown at the right side of the page.

There can be more than one assembler format for each architecture level. Floating point operations on formatted data show an assembly format with the actual assembler mnemonic for each valid value of the *fmt* field. For example, the ADD fmt instruction lists both ADD.S and ADD.D.

The assembler format lines sometimes include parenthetical comments to help explain variations in the formats (once again, see C.cond.fmt). These comments are not a part of the assembler format.

### 2.1.4 Purpose Field

The *Purpose* field gives a short description of the use of the instruction.

**Figure 2.5 Example of Instruction Purpose**

**Purpose:** Add Word

To add 32-bit integers. If an overflow occurs, then trap.

### 2.1.5 Description Field

If a one-line symbolic description of the instruction is feasible, it appears immediately to the right of the *Description* heading. The main purpose is to show how fields in the instruction are used in the arithmetic or logical operation.

**Figure 2.6 Example of Instruction Description**

**Description:** `GPR[rd] ← GPR[rs] + GPR[rt]`

The 32-bit word value in GPR *rt* is added to the 32-bit value in GPR *rs* to produce a 32-bit result.

- If the addition results in 32-bit 2's complement arithmetic overflow, the destination register is not modified and an Integer Overflow exception occurs.
- If the addition does not overflow, the 32-bit result is signed-extended and placed into GPR *rd*.

The body of the section is a description of the operation of the instruction in text, tables, and figures. This description complements the high-level language description in the *Operation* section.

This section uses acronyms for register descriptions. "GPR *rt*" is CPU general-purpose register specified by the instruction field *rt*. "FPR *fs*" is the floating point operand register specified by the instruction field *fs*. "CP1 register *fd*" is the coprocessor 1 general register specified by the instruction field *fd*. "*FCSR*" is the floating point *Control / Status* register.

### 2.1.6 Restrictions Field

The *Restrictions* field documents any possible restrictions that may affect the instruction. Most restrictions fall into one of the following six categories:

- Valid values for instruction fields (for example, see floating point ADD fmt)
- ALIGNMENT requirements for memory addresses (for example, see LW)
- Valid values of operands (for example, see DADD)
- Valid operand formats (for example, see floating point ADD.fmt)

- Order of instructions necessary to guarantee correct execution. These ordering constraints avoid pipeline hazards for which some processors do not have hardware interlocks (for example, see MUL).
- Valid memory access types (for example, see LL/SC)

**Figure 2.7 Example of Instruction Restrictions**

**Restrictions:**

If either GPR *rt* or GPR *rs* does not contain sign-extended 32-bit values (bits $_{63..31}$ equal), then the result of the operation is `UNPREDICTABLE`.

### 2.1.7 Operation Field

The *Operation* field describes the operation of the instruction as pseudocode in a high-level language notation resembling Pascal. This formal description complements the *Description* section; it is not complete in itself because many of the restrictions are either difficult to include in the pseudocode or are omitted for legibility.

**Figure 2.8 Example of Instruction Operation**

**Operation:**

```
if NotWordValue(GPR[rs]) or NotWordValue(GPR[rt]) then
   UNPREDICTABLE
endif
temp ← (GPR[rs]31||GPR[rs]31..0) + (GPR[rt]31||GPR[rt]31..0)
if temp32 ≠ temp31 then
   SignalException(IntegerOverflow)
else
   GPR[rd] ← sign_extend(temp31..0)
endif
```

See 2.2 "Operation Section Notation and Functions" on page 23 for more information on the formal notation used here.

### 2.1.8 Exceptions Field

The *Exceptions* field lists the exceptions that can be caused by *Operation* of the instruction. It omits exceptions that can be caused by the instruction fetch, for instance, TLB Refill, and also omits exceptions that can be caused by asynchronous external events such as an Interrupt. Although a Bus Error exception may be caused by the operation of a load or store instruction, this section does not list Bus Error for load and store instructions because the relationship between load and store instructions and external error indications, like Bus Error, are dependent upon the implementation.

**Figure 2.9 Example of Instruction Exception**

**Exceptions:**

Integer Overflow

An instruction may cause implementation-dependent exceptions that are not present in the *Exceptions* section.

### 2.1.9 Programming Notes and Implementation Notes Fields

The *Notes* sections contain material that is useful for programmers and implementors, respectively, but that is not necessary to describe the instruction and does not belong in the description sections.

**Figure 2.10 Example of Instruction Programming Notes**

**Programming Notes:**

ADDU performs the same arithmetic operation but does not trap on overflow.

## 2.2 Operation Section Notation and Functions

In an instruction description, the *Operation* section uses a high-level language notation to describe the operation performed by each instruction. Special symbols used in the pseudocode are described in the previous chapter. Specific pseudocode functions are described below.

This section presents information about the following topics:


- "Instruction Execution Ordering" on page 23
- "Pseudocode Functions" on page 23


### 2.2.1 Instruction Execution Ordering

Each of the high-level language statements in the *Operations* section are executed sequentially (except as constrained by conditional and loop constructs).

### 2.2.2 Pseudocode Functions

There are several functions used in the pseudocode descriptions. These are used either to make the pseudocode more readable, to abstract implementation-specific behavior, or both. These functions are defined in this section, and include the following:


- "Coprocessor General Register Access Functions" on page 23
- "Memory Operation Functions" on page 25
- "Floating Point Functions" on page 28
- "Miscellaneous Functions" on page 31


#### 2.2.2.1 Coprocessor General Register Access Functions

Defined coprocessors, except for CP0, have instructions to exchange words and doublewords between coprocessor general registers and the rest of the system. What a coprocessor does with a word or doubleword supplied to it and how a coprocessor supplies a word or doubleword is defined by the coprocessor itself. This behavior is abstracted into the functions described in this section.

### *COP_LW*

The COP_LW function defines the action taken by coprocessor z when supplied with a word from memory during a load word operation. The action is coprocessor-specific. The typical action would be to store the contents of memword in coprocessor general register *rt*.

**Figure 2.11 COP_LW Pseudocode Function**

```
COP_LW (z, rt, memword)
    z: The coprocessor unit number
    rt: Coprocessor general register specifier
    memword: A 32-bit word value supplied to the coprocessor

    /* Coprocessor-dependent action */

endfunction COP_LW
```

### *COP_LD*

The COP_LD function defines the action taken by coprocessor z when supplied with a doubleword from memory during a load doubleword operation. The action is coprocessor-specific. The typical action would be to store the contents of memdouble in coprocessor general register *rt*.

**Figure 2.12 COP_LD Pseudocode Function**

```
COP_LD (z, rt, memdouble)
    z: The coprocessor unit number
    rt: Coprocessor general register specifier
    memdouble:  64-bit doubleword value supplied to the coprocessor.

    /* Coprocessor-dependent action */

endfunction COP_LD
```

### *COP_SW*

The COP_SW function defines the action taken by coprocessor *z* to supply a word of data during a store word operation. The action is coprocessor-specific. The typical action would be to supply the contents of the low-order word in coprocessor general register *rt*.

**Figure 2.13 COP_SW Pseudocode Function**

```
dataword ← COP_SW (z, rt)
    z: The coprocessor unit number
    rt: Coprocessor general register specifier
    dataword: 32-bit word value

    /* Coprocessor-dependent action */

endfunction COP_SW
```

### *COP_SD*

The COP_SD function defines the action taken by coprocessor *z* to supply a doubleword of data during a store doubleword operation. The action is coprocessor-specific. The typical action would be to supply the contents of the low-order doubleword in coprocessor general register *rt*.

**Figure 2.14 COP_SD Pseudocode Function**

```
datadouble ← COP_SD (z, rt)
    z: The coprocessor unit number
    rt: Coprocessor general register specifier
    datadouble: 64-bit doubleword value

    /* Coprocessor-dependent action */

endfunction COP_SD
```

#### *CoprocessorOperation*

The CoprocessorOperation function performs the specified Coprocessor operation.

**Figure 2.15 CoprocessorOperation Pseudocode Function**

```
CoprocessorOperation (z, cop_fun)

    /* z:         Coprocessor unit number */
    /* cop_fun:   Coprocessor function from function field of instruction */

    /* Transmit the cop_fun value to coprocessor z */

endfunction CoprocessorOperation
```

### 2.2.2.2 Memory Operation Functions

Regardless of byte ordering (big- or little-endian), the address of a halfword, word, or doubleword is the smallest byte address of the bytes that form the object. For big-endian ordering this is the most-significant byte; for a little-endian ordering this is the least-significant byte.

In the *Operation* pseudocode for load and store operations, the following functions summarize the handling of virtual addresses and the access of physical memory. The size of the data item to be loaded or stored is passed in the *AccessLength* field. The valid constant names and values are shown in Table 2.1. The bytes within the addressed unit of memory (word for 32-bit processors or doubleword for 64-bit processors) that are used can be determined directly from the *AccessLength* and the two or three low-order bits of the address.

#### *AddressTranslation*

The AddressTranslation function translates a virtual address to a physical address and its cacheability and coherency attribute, describing the mechanism used to resolve the memory reference.

Given the virtual address *vAddr*, and whether the reference is to Instructions or Data (*IorD*), find the corresponding physical address (*pAddr*) and the cacheability and coherency attribute (*CCA*) used to resolve the reference. If the virtual address is in one of the unmapped address spaces, the physical address and *CCA* are determined directly by the virtual address. If the virtual address is in one of the mapped address spaces then the TLB or fixed mapping MMU determines the physical address and access type; if the required translation is not present in the TLB or the desired access is not permitted, the function fails and an exception is taken.

**Figure 2.16 AddressTranslation Pseudocode Function**

```
(pAddr, CCA) ← AddressTranslation (vAddr, IorD, LorS)

    /* pAddr: physical address */
    /* CCA:   Cacheability&Coherency Attribute,the method used to access caches*/
```

```
    /*        and memory and resolve the reference */

    /* vAddr: virtual address */
    /* IorD:  Indicates whether access is for INSTRUCTION or DATA */
    /* LorS:  Indicates whether access is for LOAD or STORE */

    /* See the address translation description for the appropriate MMU */
    /* type in Volume III of this book for the exact translation mechanism */

endfunction AddressTranslation
```

### *LoadMemory*

The LoadMemory function loads a value from memory.

This action uses cache and main memory as specified in both the Cacheability and Coherency Attribute (*CCA*) and the access (*IorD*) to find the contents of *AccessLength* memory bytes, starting at physical location *pAddr*. The data is returned in a fixed-width naturally aligned memory element (*MemElem*). The low-order 2 (or 3) bits of the address and the *AccessLength* indicate which of the bytes within *MemElem* need to be passed to the processor. If the memory access type of the reference is *uncached*, only the referenced bytes are read from memory and marked as valid within the memory element. If the access type is *cached* but the data is not present in cache, an implementation-specific *size* and *alignment* block of memory is read and loaded into the cache to satisfy a load reference. At a minimum, this block is the entire memory element.

**Figure 2.17 LoadMemory Pseudocode Function**

```
MemElem ← LoadMemory (CCA, AccessLength, pAddr, vAddr, IorD)

    /* MemElem:  Data is returned in a fixed width with a natural alignment. The */
    /*           width is the same size as the CPU general-purpose register, */
    /*           32 or 64 bits, aligned on a 32- or 64-bit boundary, */
    /*           respectively. */
    /* CCA:      Cacheability&CoherencyAttribute=method used to access caches */
    /*           and memory and resolve the reference */

    /* AccessLength: Length, in bytes, of access */
    /* pAddr:    physical address */
    /* vAddr:    virtual address */
    /* IorD:     Indicates whether access is for Instructions or Data */

endfunction LoadMemory
```

### *StoreMemory*

The StoreMemory function stores a value to memory.

The specified data is stored into the physical location *pAddr* using the memory hierarchy (data caches and main memory) as specified by the Cacheability and Coherency Attribute (*CCA*). The *MemElem* contains the data for an aligned, fixed-width memory element (a word for 32-bit processors, a doubleword for 64-bit processors), though only the bytes that are actually stored to memory need be valid. The low-order two (or three) bits of *pAddr* and the *AccessLength* field indicate which of the bytes within the *MemElem* data should be stored; only these bytes in memory will actually be changed.

**Figure 2.18 StoreMemory Pseudocode Function**

```
StoreMemory (CCA, AccessLength, MemElem, pAddr, vAddr)
```

```
    /* CCA:       Cacheability&Coherency Attribute, the method used to access */
    /*            caches and memory and resolve the reference. */
    /* AccessLength: Length, in bytes, of access */
    /* MemElem:   Data in the width and alignment of a memory element. */
    /*            The width is the same size as the CPU general */
    /*            purpose register, either 4 or 8 bytes, */
    /*            aligned on a 4- or 8-byte boundary. For a */
    /*            partial-memory-element store, only the bytes that will be*/
    /*            stored must be valid.*/
    /* pAddr:     physical address */
    /* vAddr:     virtual address */

endfunction StoreMemory
```

### *Prefetch*

The Prefetch function prefetches data from memory.

Prefetch is an advisory instruction for which an implementation-specific action is taken. The action taken may increase performance but must not change the meaning of the program or alter architecturally visible state.

**Figure 2.19 Prefetch Pseudocode Function**

```
Prefetch (CCA, pAddr, vAddr, DATA, hint)

    /* CCA:    Cacheability&Coherency Attribute, the method used to access */
    /*         caches and memory and resolve the reference. */
    /* pAddr: physical address */
    /* vAddr: virtual address */
    /* DATA:  Indicates that access is for DATA */
    /* hint:  hint that indicates the possible use of the data */

endfunction Prefetch
```

Table 2.1 lists the data access lengths and their labels for loads and stores.

**Table 2.1 AccessLength Specifications for Loads/Stores**

| AccessLength Name | Value | Meaning |
|---|---|---|
| DOUBLEWORD | 7 | 8 bytes (64 bits) |
| SEPTIBYTE | 6 | 7 bytes (56 bits) |
| SEXTIBYTE | 5 | 6 bytes (48 bits) |
| QUINTIBYTE | 4 | 5 bytes (40 bits) |
| WORD | 3 | 4 bytes (32 bits) |
| TRIPLEBYTE | 2 | 3 bytes (24 bits) |
| HALFWORD | 1 | 2 bytes (16 bits) |
| BYTE | 0 | 1 byte (8 bits) |

### *SyncOperation*

The SyncOperation function orders loads and stores to synchronize shared memory.

This action makes the effects of the synchronizable loads and stores indicated by *stype* occur in the same order for all processors.

**Figure 2.20 SyncOperation Pseudocode Function**

```
SyncOperation(stype)

   /* stype: Type of load/store ordering to perform. */

   /* Perform implementation-dependent operation to complete the */
   /* required synchronization operation */

endfunction SyncOperation
```

### 2.2.2.3 Floating Point Functions

The pseudocode shown in below specifies how the unformatted contents loaded or moved to CP1 registers are interpreted to form a formatted value. If an FPR contains a value in some format, rather than unformatted contents from a load (uninterpreted), it is valid to interpret the value in that format (but not to interpret it in a different format).

#### *ValueFPR*

The ValueFPR function returns a formatted value from the floating point registers.

**Figure 2.21 ValueFPR Pseudocode Function**

```
value ← ValueFPR(fpr, fmt)

   /* value: The formattted value from the FPR */

   /* fpr:   The FPR number */
   /* fmt:   The format of the data, one of: */
   /*        S, D, W, L, PS, */
   /*        OB, QH, */
   /*        UNINTERPRETED_WORD, */
   /*        UNINTERPRETED_DOUBLEWORD */
   /* The UNINTERPRETED values are used to indicate that the datatype */
   /* is not known as, for example, in SWC1 and SDC1 */

   case fmt of
      S, W, UNINTERPRETED_WORD:
         valueFPR ← UNPREDICTABLE^32 || FPR[fpr]_31..0

      D, UNINTERPRETED_DOUBLEWORD:
         if (FP32RegistersMode = 0)
            if (fpr_0 ≠ 0) then
               valueFPR ← UNPREDICTABLE
            else
               valueFPR ← FPR[fpr+1]_31..0 || FPR[fpr]_31..0
            endif
         else
            valueFPR ← FPR[fpr]
         endif

      L, PS, OB, QH:
         if (FP32RegistersMode = 0) then
            valueFPR ← UNPREDICTABLE
```

```
            else
                valueFPR ← FPR[fpr]
            endif

        DEFAULT:
            valueFPR ← UNPREDICTABLE

    endcase
endfunction ValueFPR
```

The pseudocode shown below specifies the way a binary encoding representing a formatted value is stored into CP1 registers by a computational or move operation. This binary representation is visible to store or move-from instructions. Once an FPR receives a value from the StoreFPR(), it is not valid to interpret the value with ValueFPR() in a different format.

### *StoreFPR*

**Figure 2.22 StoreFPR Pseudocode Function**

```
StoreFPR (fpr, fmt, value)

    /* fpr:   The FPR number */
    /* fmt:   The format of the data, one of: */
    /*        S, D, W, L, PS, */
    /*        OB, QH, */
    /*        UNINTERPRETED_WORD, */
    /*        UNINTERPRETED_DOUBLEWORD */
    /* value: The formattted value to be stored into the FPR */

    /* The UNINTERPRETED values are used to indicate that the datatype */
    /* is not known as, for example, in LWC1 and LDC1 */


    case fmt of
        S, W, UNINTERPRETED_WORD:
            FPR[fpr] ← UNPREDICTABLE^32 || value_31..0

        D, UNINTERPRETED_DOUBLEWORD:
            if (FP32RegistersMode = 0)
                if (fpr_0 ≠ 0) then
                    UNPREDICTABLE
                else
                    FPR[fpr]   ← UNPREDICTABLE^32 || value_31..0
                    FPR[fpr+1] ← UNPREDICTABLE^32 || value_63..32
                endif
            else
                FPR[fpr] ← value
            endif

        L, PS, OB, QH:
            if (FP32RegistersMode = 0) then
                UNPREDICTABLE
            else
                FPR[fpr] ← value
            endif

    endcase
```

```
endfunction StoreFPR
```

The pseudocode shown below checks for an enabled floating point exception and conditionally signals the exception.

### *CheckFPException*

**Figure 2.23 CheckFPException Pseudocode Function**

```
CheckFPException()

/* A floating point exception is signaled if the E bit of the Cause field is a 1 */
/* (Unimplemented Operations have no enable) or if any bit in the Cause field */
/* and the corresponding bit in the Enable field are both 1 */

   if (   (FCSR17 = 1) or
          ((FCSR16..12 and FCSR11..7) ≠ 0))   ) then
      SignalException(FloatingPointException)
   endif

endfunction CheckFPException
```

### *FPConditionCode*

The FPConditionCode function returns the value of a specific floating point condition code.

**Figure 2.24 FPConditionCode Pseudocode Function**

```
tf ←FPConditionCode(cc)

   /* tf: The value of the specified condition code */

   /* cc: The Condition code number in the range 0..7 */

   if cc = 0 then
      FPConditionCode ← FCSR23
   else
      FPConditionCode ← FCSR24+cc
   endif

endfunction FPConditionCode
```

### *SetFPConditionCode*

The SetFPConditionCode function writes a new value to a specific floating point condition code.

**Figure 2.25 SetFPConditionCode Pseudocode Function**

```
SetFPConditionCode(cc, tf)
   if cc = 0 then
      FCSR ← FCSR31..24 || tf || FCSR22..0
   else
      FCSR ← FCSR31..25+cc || tf || FCSR23+cc..0
endif

endfunction SetFPConditionCode
```

#### 2.2.2.4 Miscellaneous Functions

This section lists miscellaneous functions not covered in previous sections.

***SignalException***

The SignalException function signals an exception condition.

This action results in an exception that aborts the instruction. The instruction operation pseudocode never sees a return from this function call.

**Figure 2.26 SignalException Pseudocode Function**

```
SignalException(Exception, argument)

   /* Exception:    The exception condition that exists. */
   /* argument:     A exception-dependent argument, if any */

endfunction SignalException
```

***SignalDebugBreakpointException***

The SignalDebugBreakpointException function signals a condition that causes entry into Debug Mode from non-Debug Mode.

This action results in an exception that aborts the instruction. The instruction operation pseudocode never sees a return from this function call.

**Figure 2.27 SignalDebugBreakpointException Pseudocode Function**

```
SignalDebugBreakpointException()

endfunction SignalDebugBreakpointException
```

***SignalDebugModeBreakpointException***

The SignalDebugModeBreakpointException function signals a condition that causes entry into Debug Mode from Debug Mode (i.e., an exception generated while already running in Debug Mode).

This action results in an exception that aborts the instruction. The instruction operation pseudocode never sees a return from this function call.

**Figure 2.28 SignalDebugModeBreakpointException Pseudocode Function**

```
SignalDebugModeBreakpointException()

endfunction SignalDebugModeBreakpointException
```

***NullifyCurrentInstruction***

The NullifyCurrentInstruction function nullifies the current instruction.

The instruction is aborted, inhibiting not only the functional effect of the instruction, but also inhibiting all exceptions detected during fetch, decode, or execution of the instruction in question. For branch-likely instructions, nullification kills the instruction in the delay slot of the branch likely instruction.

**Figure 2.29 NullifyCurrentInstruction PseudoCode Function**

```
NullifyCurrentInstruction()

endfunction NullifyCurrentInstruction
```

### *JumpDelaySlot*

The JumpDelaySlot function is used in the pseudocode for the PC-relative instructions in the MIPS16e ASE. The function returns TRUE if the instruction at *vAddr* is executed in a jump delay slot. A jump delay slot always immediately follows a JR, JAL, JALR, or JALX instruction.

**Figure 2.30 JumpDelaySlot Pseudocode Function**

```
JumpDelaySlot(vAddr)

   /* vAddr:Virtual address */

endfunction JumpDelaySlot
```

### *NotWordValue*

The NotWordValue function returns a boolean value that determines whether the 64-bit value contains a valid word (32-bit) value. Such a value has bits 63..32 equal to bit 31.

**Figure 2.31 NotWordValue Pseudocode Function**

```
result ← NotWordValue(value)

   /* result:   True if the value is not a correct sign-extended word value; */
   /*           False otherwise */

   /* value:    A 64-bit register value to be checked */

   NotWordValue ← value63..32 ≠ (value31)32

endfunction NotWordValue
```

### *PolyMult*

The PolyMult function multiplies two binary polynomial coefficients.

**Figure 2.32 PolyMult Pseudocode Function**

```
PolyMult(x, y)
   temp ← 0
   for i in 0 .. 31
      if xi = 1 then
         temp ← temp xor (y(31-i)..0 || 0i)
      endif
   endfor

   PolyMult ← temp

endfunction PolyMult
```

## 2.3 Op and Function Subfield Notation

In some instructions, the instruction subfields *op* and *function* can have constant 5- or 6-bit values. When reference is made to these instructions, uppercase mnemonics are used. For instance, in the floating point ADD instruction, *op*=COP1 and *function*=ADD. In other cases, a single field has both fixed and variable subfields, so the name contains both upper- and lowercase characters.

## 2.4 FPU Instructions

In the detailed description of each FPU instruction, all variable subfields in an instruction format (such as *fs, ft, immediate*, and so on) are shown in lowercase. The instruction name (such as ADD, SUB, and so on) is shown in uppercase.

For the sake of clarity, an alias is sometimes used for a variable subfield in the formats of specific instructions. For example, *rs=base* in the format for load and store instructions. Such an alias is always lowercase since it refers to a variable subfield.

Bit encodings for mnemonics are given in Volume I, in the chapters describing the CPU, FPU, MDMX, and MIPS16e instructions.

See "Op and Function Subfield Notation" on page 33 for a description of the *op* and *function* subfields.

*Chapter 3*

# The MIPS64® SIMD Architecture

The MIPS® SIMD Architecture (MSA) module adds new instructions to the industry-standard MIPS Release 5 ("R5") architecture that allow efficient parallel processing of vector operations. This functionality is of growing importance across a range of consumer electronics and enterprise applications.

In consumer electronics, while dedicated, non-programmable hardware aids the CPU and GPU by handling heavy-duty multimedia codecs, there is a recognized trend toward adding a software-programmable solution in the CPU to handle emerging applications or a small number of functions not covered by the dedicated hardware. In this way, SIMD can provide increased system flexibility, and the MSA is ideal for these applications.

However, the MSA is not just another multimedia SIMD extension. Rather than focusing on narrowly defined instructions that must have optimized code written manually in assembly language in order to be utilized, the MSA is designed to accelerate compute-intensive applications in conjunction with leveraging generic compiler support.

A wide range of applications – including data mining, feature extraction in video, image and video processing, human-computer interaction, and others – have some built-in data parallelism that lends itself well to SIMD. These compute-intensive software packages will not be written in assembly for any specific architecture, but rather in high-level languages using operations on vector data types.

The MSA module was implemented with strict adherence to RISC (Reduced Instruction Set Computer) design principles. From the beginning, MIPS architects designed the MSA with a carefully selected, simple SIMD instruction set that is not only programmer- and compiler-friendly, but also hardware-efficient in terms of speed, area, and power consumption. The simple instructions are also easy to support within high-level languages, enabling fast and simple development of new code, as well as leverage of existing code.

This chapter describes the purpose and key features of the MIPS64® SIMD Architecture (MSA).

## 3.1 Overview

The MSA complements the well-established MIPS architecture with a set of more than 150 new instructions operating on 32 vector registers of 8-, 16-, 32-, and 64-bit integer, 16-and 32-bit fixed- point, or 32- and 64-bit floating-point data elements. In the current release, MSA implements 128-bit wide vector registers shared with the 64-bit wide floating-point unit (FPU) registers.

In multi-threaded implementations, MSA allows for fewer than 32 physical vector registers per hardware thread context. The thread contexts have access to as many vector registers as needed, up to the full 32 vector registers set defined by the architecture. When the hardware runs out of physical vector registers, the OS re-schedules the running threads or processes to accommodate the pending requests. The actual mapping of the physical vector registers to the hardware thread contexts is managed by the hardware.

The MSA floating-point implementation is compliant with the IEEE Standard for Floating-Point Arithmetic 754™-2008. All standard operations are provided for 32-bit and 64-bit floating-point data. 16-bit floating-point storage format is supported through conversion instructions to/from 32-bit floating-point data. In the case of a float-

ing-point exception, each faulting vector element is precisely identified without the need for software emulation for all vector elements.

For compare and branch, MSA uses no global condition flags: compare instructions write the results per vector element as all zero or all one bit values. Branch instructions test for zero or not zero element(s) or vector value.

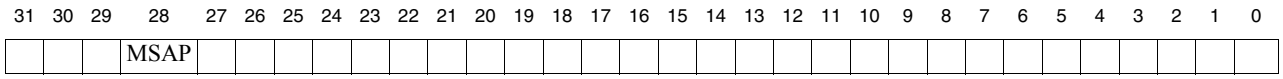
MSA is built on the same principles pioneered by MIPS and its earlier MDMX (MIPS Digital Media eXtension): a simple, yet very efficient instruction set. The opcodes allocated to MDMX are reused for MSA, which means that MDMX is deprecated at the time of the release of MSA.

MSA requires a compliant implementation of the MIPS32 Architecture, Release 5 or later.

## 3.2 MSA Software Detection

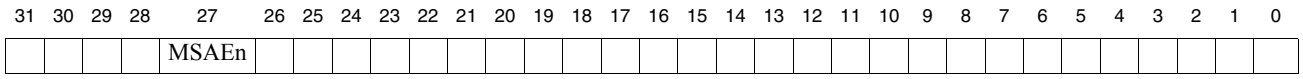
The presence of MSA implementation is indicated by the *Config3* MSAP bit (CP0 Register 16, Select 3, bit 28) as shown in Figure 3-1. MSAP bit is fixed by the hardware implementation and is read-only for the software. The software may determine if the MSA is implemented by checking if the MSAP bit is set. Any attempt to execute MSA instructions must cause a Reserved Instruction Exception if the MSAP bit is not set.

**Figure 3-1 Config3 (CP0 Register 16, Select 3) MSA Implementation Present Bit**




*Config5* MSAEn bit (CP0 Register 16, Select 5, bit 27), shown in Figure 3-2, is used to enable the MSA instructions. Executing a MSA instruction when MSAEn bit is not set causes a MSA Disabled Exception, see Section 3.5.1 "Handling the MSA Disabled Exception". The reset state of the MSAEn bit is zero.

**Figure 3-2 Config5 (CP0 Register 16, Select 5) MSA Enable Bit**




## 3.3 MSA Vector Registers

The MSA operates on 32 128-bit wide vector registers. If both MSA and the scalar floating-point unit (FPU) are present, the 128-bit MSA vector registers extend and share the 64-bit FPU registers. MSA and FPU can not be both present, unless the FPU has 64-bit floating-point registers.

MSA vector register have four data formats: byte (8-bit), halfword (16-bit), word (32-bit), doubleword (64-bit). Corresponding to the associated data format, a vector register consists of a number of elements indexed from 0 to n,

where the least significant bit of the $0^{th}$ element is the vector register bit 0 and the most significant bit of the $n^{th}$ element is the vector register bit 127.

When both FPU and MSA are present, the floating-point registers are mapped on the corresponding MSA vector registers as the $0^{th}$ elements.

### 3.3.1 Registers Layout

Figure 3-3 through Figure 3-6 show the vector register layout for elements of all four data formats where [n] refers to the $n^{th}$ vector element and MSB and LSB stand for the element's Most Significant and Least Significant Byte.

**Figure 3-3 MSA Vector Register Byte Elements**

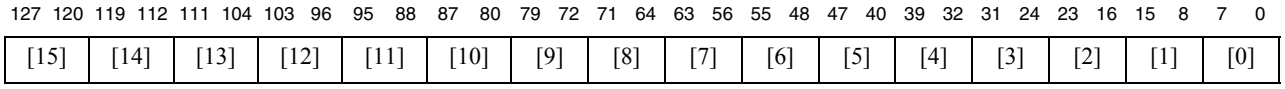



**Figure 3-4 MSA Vector Register Halfword Elements**

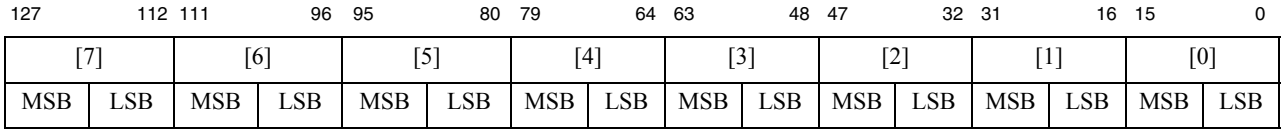



**Figure 3-5 MSA Vector Register Word Elements**

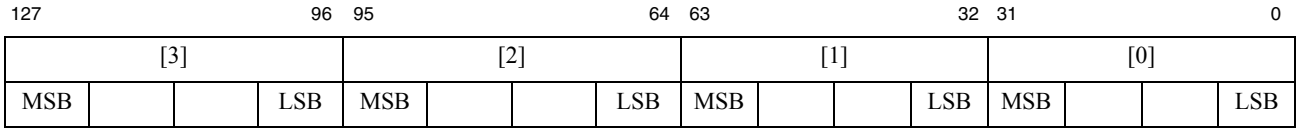
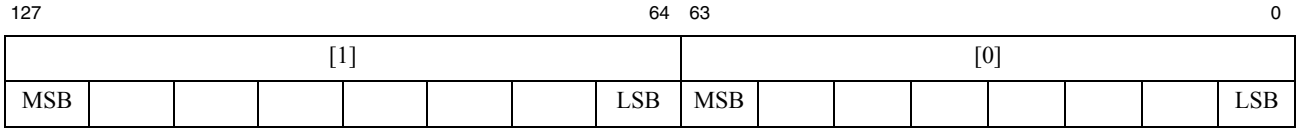



**Figure 3-6 MSA Vector Register Doubleword Elements**



The vector register layout for slide instructions SLD and SLDI is a 2-dimensional byte array, with as many rows as bytes in the integer data format. For byte data format, the 1-row array is reduced to the vector shown in Figure 3-3. For halfword, the byte array has 2 rows (Figure 3-7), there are 4 rows for word (Figure 3-8), and 8 rows (Figure 3-9) for doubleword data format.

**Figure 3-7 MSA Vector Register as 2-Row Byte Array**

| 63 56 | 55 48 | 47 40 | 39 32 | 31 24 | 23 16 | 15 8 | 7 0 |
|---|---|---|---|---|---|---|---|
| [15] | [14] | [13] | [12] | [11] | [10] | [9] | [8] |
| [7] | [6] | [5] | [4] | [3] | [2] | [1] | [0] |

**Figure 3-8 MSA Vector Register as 4-Row Byte Array**

| 31 24 | 23 16 | 15 8 | 7 0 |
|---|---|---|---|
| [15] | [14] | [13] | [12] |
| [11] | [10] | [9] | [8] |
| [7] | [6] | [5] | [4] |
| [3] | [2] | [1] | [0] |

**Figure 3-9 MSA Vector Register as 8-Row Byte Array**

| 15 8 | 7 0 |
|---|---|
| [15] | [14] |
| [13] | [12] |
| [11] | [10] |
| [9] | [8] |
| [7] | [6] |
| [5] | [4] |
| [3] | [2] |
| [1] | [0] |

MSA vectors are stored in memory starting from the $0^{th}$ element at the lowest byte address. The byte order of each element follows the big- or little-endian convention as indicated by the BE bit in the CP0 *Config* register (CP0 Register 16, Select 0, bit 15). For example, Table 3.1 shows the memory representation for a MSA vector consisting of word elements in both big- and little-endian mode.

**Table 3.1 Word Vector Memory Representation**

| Word Vector Element | | Little-Endian Byte Address Offset | Big-Endian Byte Address Offset |
|---|---|---|---|
| Word [0] | Byte [0] / LSB | 0 | 3 |
| | Byte [1] | 1 | 2 |
| | Byte [2] | 2 | 1 |
| | Byte [3] / MSB | 3 | 0 |
| Word [1] | Byte [0] / LSB | 4 | 7 |
| | Byte [1] | 5 | 6 |
| | Byte [2] | 6 | 5 |
| | Byte [3] / MSB | 7 | 4 |
| Word [2] | Byte [0] / LSB | 8 | 11 |
| | Byte [1] | 9 | 10 |
| | Byte [2] | 10 | 9 |
| | Byte [3] / MSB | 11 | 8 |
| Word [3] | Byte [0] / LSB | 12 | 15 |
| | Byte [1] | 13 | 14 |
| | Byte [2] | 14 | 13 |
| | Byte [3] / MSB | 15 | 12 |

## 3.3.2 Floating-Point Registers Mapping

The scalar floating-point unit (FPU) registers are mapped on the MSA vector registers. To facilitate register data sharing between scalar floating-point instructions and vector instructions, the FPU is required to use 64-bit floating-point registers operating in 64-bit mode. More specifically:

- If MSA and FPU are both present, then the FPU must implement 64-bit floating point registers, i.e. bits $Config3_{MSAP}$ and $FIR_{F64}$ (CP1 Control Register 0, bit 22) are set.
- If MSA and FPU are both present, then the FPU must be compliant with the IEEE Standard for Floating-Point Arithmetic 754™-2008, i.e. the read-only bits $FCSR_{NAN2008}$ and $FCSR_{ABS2008}$ (CP1 Control Register 31, bits 18 and 19) are set.
- MSA instructions are not enabled while the FPU (Coprocessor 1) is usable and operates in 32-bit mode. i.e. bit $Status_{CU1}$ (CP Register 12, Select 0, bit 29) is set and bit $Status_{FR}$ (CP Register 12, Select 0, bit 26) is not set. Any attempt to execute MSA instructions with $Status_{CU1}$ set and $Status_{FR}$ clear will generate the Reserved Instruction exception.

When $Status_{FR}$ is set, the read and write operations for the FPU/MSA mapped floating-point registers are defined as follows:

- A read operation from the floating-point register *r*, where *r* = 0, …, 31, returns the value of the element with index 0 in the vector register *r*. The element's format is word for 32-bit (single precision floating-point) read or double for 64-bit (double precision floating-point) read.
- A 32-bit read operation from the high part of the floating-point register *r*, where *r* = 0, …, 31, returns the value of the word element with index 1 in the vector register *r*.
- A write operation of value *V* to the floating-point register *r*, where *r* = 0, …, 31, writes *V* to the element with index 0 in the vector register *r* and all remaining elements are **UNPREDICTABLE**. Figure 3-10 and Figure 3-11 show the vector register *r* after writing a 32-bit (single precision floating-point) and a 64-bit (double precision floating-point) value *V* to the floating-point register *r*.
- A 32-bit write operation of value *V* to the high part of the floating-point register *r*, where *r* = 0, …, 31, writes *V* to the word element with index 1 in the vector register *r*, **preserves** word element 0, and all remaining elements are **UNPREDICTABLE**. Figure 3-12 shows the vector register *r* after writing a 32-bit value *V* to the floating-point register *r*.

Changing the $Status_{FR}$ value renders all floating-point and vector registers **UNPREDICTABLE.**

**Figure 3-10 FPU Word Write Effect on the MSA Vector Register ($Status_{FR}$ set)**

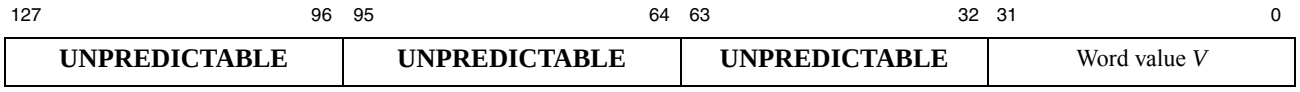



**Figure 3-11 FPU Doubleword Write Effect on the MSA Vector Register ($Status_{FR}$ set)**

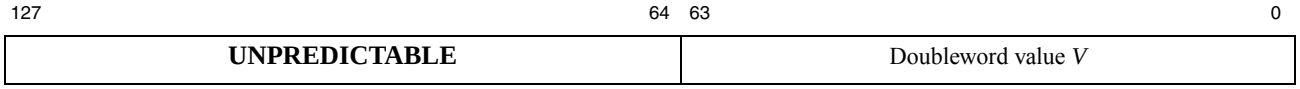
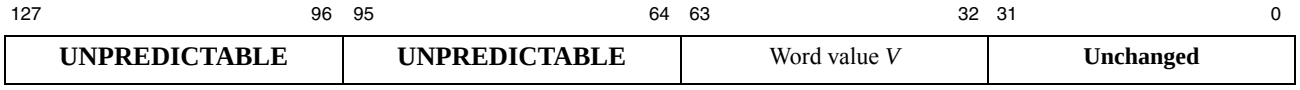



**Figure 3-12 FPU High Word Write Effect on the MSA Vector Register ($Status_{FR}$ set)**




## 3.4 MSA Control Registers

The control registers are used to record and manage the MSA state and resources. Two dedicated instructions are provided for this purpose: CFCMSA (Copy From Control MSA register) and CTCMSA (Copy To Control MSA register). The only information residing outside the MSA control registers is the implementation bit $Config3_{MSAP}$ and the

enable bit *Config5*$_{MSAEn}$ discussed in Section 3.2 "MSA Software Detection".

There are 8 MSA control registers. See Table 3.2 for a summary and the following sections for the complete description.

**Table 3.2 MSA Control Registers**

| Name | Index | Access Mode | | Read/Write | Description |
|---|---|---|---|---|---|
| | | *MSAIR*$_{WRP}$ = 1 | *MSAIR*$_{WRP}$ = 0 | | |
| MSAIR | 0 | User mode accessible, not privileged | | Read Only | Implementation |
| MSACSR | 1 | User mode accessible, not privileged | | Read/Write | Control and status |
| MSAAccess | 2 | Privileged | Reserved | Read Only | Available vector registers mask |
| MSASave | 3 | Privileged | Reserved | Read/Write | Saved vector registers mask |
| MSAModify | 4 | Privileged | Reserved | Read/Write | Modified (written) vector registers mask |
| MSARequest | 5 | Privileged | Reserved | Read Only | Requested vector registers mask |
| MSAMap | 6 | Privileged | Reserved | Read/Write | Mapping vector register index |
| MSAUnmap | 7 | Privileged | Reserved | Read/Write | Unmapping vector register index |

## 3.4.1 MSA Implementation Register (MSAIR, MSA Control Register 0)

**Compliance Level:** *Required* if MSA is implemented
**Access Mode:** *Not privileged*, user mode accessible

The MSA Implementation Register (*MSAIR*) is a 32-bit read-only register that contains information specifying the identification of MSA. Figure 3-13 shows the format of the *MSAIR*; Figure 3-14 describes the *MSAIR* fields.

The software can read the *MSAIR* using CFCMSA (Copy From Control MSA register) instruction. If the multi-threading module is present, all thread contexts share one *MSAIR* register instance.

**Figure 3-13 MSAIR Register Format**

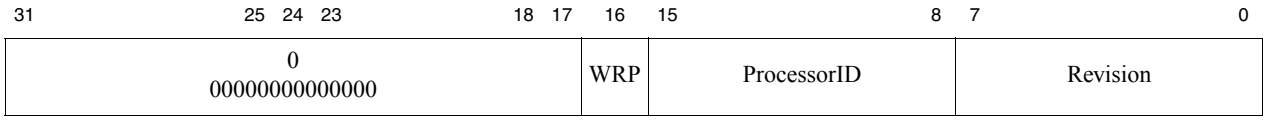


**Figure 3-14 MSAIR Register Field Descriptions**

| Fields | | Description | Read/ Write | Reset State | Compliance |
|---|---|---|---|---|---|
| Name | Bits | | | | |
| 0 | 31:17 | Reserved for future use; reads as zero and must be written as zero. | R0 | 0 | Reserved |
| WRP | 16 | Vector Registers Partitioning.<br>Using vector registers partitioning MSA allows for multithreaded implementations with fewer than 32 physical vector registers per hardware thread context.<br><br>Encoding: 0 — Vector registers partitioning not implemented.<br>Encoding: 1 — Vector registers partitioning implemented. | R | Preset | Required |
| ProcID | 15:8 | Processor ID number | R | Preset | Required |
| Rev | 7:0 | Revision number | R | Preset | Required |

## 3.4.2 MSA Control and Status Register (MSACSR, MSA Control Register 1)

**Compliance Level:** *Required* if MSA is implemented
**Access Mode:** *Not privileged*, user mode accessible

The MSA Control and Status Register (*MSACSR*) is a 32-bit read/write register that controls the operation of the MSA unit. Figure 3-15 shows the format of the *MSACSR*; Figure 3-16 describes the *MSACSR* fields.

The software can read and write the *MSACSR* using CFCMSA and CTCMSA (Copy From and To Control MSA register) instructions. If the multi-threading module is present, each thread context has its own *MSACSR* register instance.

Floating Point Control and Status Register (*FCSR*, CP1 Control Register 31) and MSA Control and Status Register (*MSACSR*) are closely related in their purpose. However, each serves a different functional unit and can exist independently of the other.

**Figure 3-15 MSACSR Register Format**

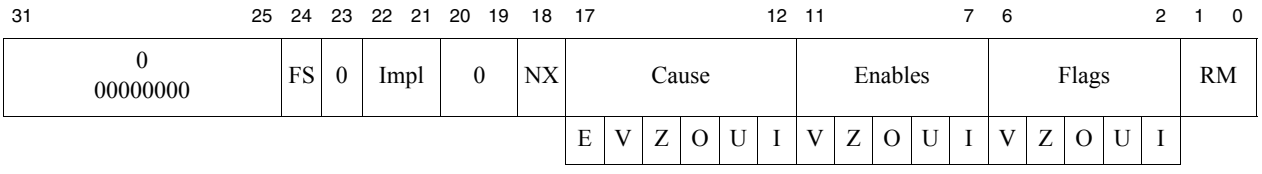


**Figure 3-16 MSACSR Register Field Descriptions**

<table>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read/ Write</th><th rowspan="2">Reset State</th><th rowspan="2">Compliance</th></tr>
<tr><th>Name</th><th>Bits</th></tr>
<tr><td>0</td><td>31:25</td><td>Reserved for future use; reads as zero and must be written as zero.</td><td>R0</td><td>0</td><td>Reserved</td></tr>
<tr><td>FS</td><td>24</td><td>Flush to zero. If not implemented, reads as zero and writes are ignored.<br>Every input subnormal value and tiny non-zero result is replaced with zero of the same sign. See Section 3.5.4 “Flush to Zero and Exception Signaling”.<br><table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>Input subnormal values and tiny non-zero results are not altered. Unimplemented Operation Exception may be signaled as needed.</td></tr><tr><td>1</td><td>Replace every input subnormal value and tiny non-zero result with zero of the same sign. No Unimplemented Operation Exception is signaled.</td></tr></table></td><td>R/W</td><td>0</td><td>Optional</td></tr>
<tr><td>0</td><td>23</td><td>Reserved for future use; reads as zero and must be written as zero.</td><td>R0</td><td>0</td><td>Reserved</td></tr>
<tr><td>Impl</td><td>22:21</td><td>Available to control implementation dependent features.</td><td>R/W</td><td>Undefined</td><td>Optional</td></tr>
<tr><td>0</td><td>20:19</td><td>Reserved for future use; reads as zero and must be written as zero.</td><td>R0</td><td>0</td><td>Reserved</td></tr>
</table>

<table>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read/ Write</th><th rowspan="2">Reset State</th><th rowspan="2">Compliance</th></tr>
<tr><th>Name</th><th>Bits</th></tr>
<tr><td>NX</td><td>18</td><td>Non-trapping floating point exception mode.<br>In normal exception mode, the destination register is not written and the floating point exceptions set the Cause bits and trap.<br>In non-trapping exception mode, the operations which would normally signal floating point exceptions do not write the Cause bits and do not trap. All the destination register's elements are set either to the calculated results or, if the operation would normally signal an exception, to signaling NaN values (see Section 3.5.2 "Handling the MSA Floating Point Exception") with the least significant 6 bits recording the specific exception type detected for that element in the same format as the Cause field. The Flags bits are updated for all floating-point operation with an IEEE exception condition that does not result in a MSA floating point exception (i.e., the Enable bit is off).<br><table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>Normal exception mode</td></tr><tr><td>1</td><td>Non-trapping exception mode</td></tr></table></td><td>R/W</td><td>0</td><td>Required for floating-point</td></tr>
<tr><td>Cause</td><td>17:12</td><td>Cause bits.<br>These bits indicate the IEEE exception conditions that arise during the execution of all operations in a vector floating-point instruction. A bit is set to 1 if the corresponding exception condition arises during the execution of any operation in the vector floating-point instruction and is set to 0 otherwise. The exception conditions caused by the preceding vector floating-point instruction can be determined by reading the Cause field.<br>Refer to Table 3.3 for the meaning of each bit.</td><td>R/W</td><td>Undefined</td><td>Required for floating-point</td></tr>
<tr><td>Enable</td><td>11:7</td><td>Enable bits.<br>These bits control whether or not a exception is taken when an IEEE exception condition arises for any of the five conditions. The exception is taken when both an Enable bit and the corresponding Cause bit are set either during the execution of any operation in vector floating-point instruction or by moving a value to MSACSR or one of its alternative representations. Note that Cause bit E (Unimplemented Operation) has no corresponding Enable bit; the non-IEEE Unimplemented Operation Exception is defined by MIPS as always enabled.<br>Refer to Table 3.3 for the meaning of each bit.</td><td>R/W</td><td>Undefined</td><td>Required for floating-point</td></tr>
</table>

| Fields | | Description | Read/ Write | Reset State | Compliance |
|---|---|---|---|---|---|
| Name | Bits | | | | |
| Flags | 6:2 | Flag bits.<br>This field shows any exception conditions that have occurred for all operations in the vector floating-point instructions completed since the flag was last reset by software. When a floating-point operation raises an IEEE exception condition that does not result in a MSA floating point exception (i.e., the Enable bit is off), the corresponding bit(s) in the Flags field are set, while the others remain unchanged. Arithmetic operations that result in a floating point exception (i.e., the Enable bit is on) do not update the Flags bits.This field is never reset by hardware and must be explicitly reset by software.<br>Refer to Table 3.3 for the meaning of each bit. | R/W | Undefined | Required for floating-point |
| RM | 1:0 | Rounding Mode.<br>This field indicates the rounding mode used for most floating point operations (some operations use a specific rounding mode).<br>Refer to Table 3.4 for the meaning of the encodings of this field. | R/W | 0 | Required for floating-point |

## Table 3.3 Cause, Enable, and Flag Bit Definitions

| Bit Name | Bit Meaning |
|---|---|
| E | Unimplemented Operation.<br>This bit exists only in the Cause field. |
| V | Invalid Operation.<br>The Invalid Operation Exception is signaled if and only if there is no usefully definable result. In these cases the operands are invalid for the operation to be performed.<br>Under default exception handling, i.e. when the Invalid Operation Exception is not enabled, the default floating-point result is a quiet NaN (see Table 3.6). |
| Z | Divide by Zero.<br>The Divide by Zero Exception is signaled if and only if an exact infinite result is defined for an operation on finite operands.<br>Under default exception handling, i.e. when the Divide by Zero Exception is not enabled, the default result is an infinity correctly signed according to the operation (see Table 3.6). |
| O | Overflow.<br>The Overflow Exception is signaled if and only if the destination format's largest finite number is exceeded in magnitude by what would have been the rounded floating-point result were the exponent range unbounded.<br>Under default exception handling, i.e. when the Overflow Exception is not enabled, the overflowed rounded result (see Table 3.6) is delivered to the destination. In addition, the Inexact bit in the Cause field is set. |

**Table 3.3 Cause, Enable, and Flag Bit Definitions**

| Bit Name | Bit Meaning |
|---|---|
| U | Underflow.<br>If enabled, the Underflow Exception is signaled when a tiny non-zero result is detected after rounding regardless of whether the rounded result is exact or inexact.<br>Under default exception handling, i.e. when the Underflow Exception is not enabled, the rounded result (see Table 3.6) is delivered to the destination and:<br>• If the rounded result is inexact, the Inexact bit in the Cause field is set.<br>• If the rounded result is exact, no bit in the Flags field is set. Such an underflow condition has no observable effect under default handling. |
| I | Inexact.<br>Unless stated otherwise, if the rounded result of an operation is inexact -- that is, it differs from what would have been computed were both exponent range and precision unbounded -- then the Inexact Exception is be signaled.<br>Under default exception handling, i.e. when the Inexact Exception is not enabled, the rounded result is delivered to the destination (see Table 3.6). |

**Table 3.4 Rounding Modes Definitions**

| RM Field Encoding | Meaning |
|---|---|
| 0 | Round to nearest / ties to even.<br>Rounds the result to the nearest representable value. When two representable values are equally near, the result is rounded to the value whose least significant bit is zero (that is, even) |
| 1 | Round toward zero.<br>Rounds the result to the value closest to but not greater in magnitude than the result. |
| 2 | Round towards positive / plus infinity.<br>Rounds the result to the value closest to but not less than the result. |
| 3 | Round towards negative / minus infinity.<br>Rounds the result to the value closest to but not greater than the result. |

### 3.4.3 MSA Access Register (MSAAccess, MSA Control Register 2)

**Compliance Level:** *Required* for vector registers partitioning (i.e. $MSAIR_{WRP}$ set), otherwise *Reserved*
**Access Mode:** *Privileged*, accessible only when access to Coprocessor 0 is enabled

The MSA Access register (*MSAAccess*) is a 32-bit read-only register specifying which of the 32 architecturally defined vector registers W0, …, W31 are available to the software. Figure 3-17 shows the format of the *MSAAccess*. Vector register W*n*, where $n = 0, \ldots, 31$, is available and can be used only if $MSAAccess_{Wn}$ bit is set. The reset state of the MSA Access register is zero.

The software can read the *MSAAccess* using CFCMSA (Copy From Control MSA register) instruction. If the multi-threading module is present, each thread context has its own *MSAAccess* register instance.

To get access to vector register W*n*, *n* = 0, …, 31, the software writes *n* to *MSAMap*. W*n* is mapped to an available physical register and $MSAAccess_{Wn}$ is set. To free up an already mapped vector register W*n*, the software writes *n* to *MSAUnmap*. W*n* is unmapped and $MSAAccess_{Wn}$ cleared.

The total number of vector registers mapped at any time can not exceed the number of physical registers implemented.

**Figure 3-17 MSAAccess Register Format**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| W31 | W30 | W29 | W28 | W27 | W26 | W25 | W24 | W23 | W22 | W21 | W20 | W19 | W18 | W17 | W16 | W15 | W14 | W13 | W12 | W11 | W10 | W9 | W8 | W7 | W6 | W5 | W4 | W3 | W2 | W1 | W0 |

## 3.4.4 MSA Save Register (MSASave, MSA Control Register 3)

**Compliance Level:** *Required* for vector registers partitioning (i.e. $MSAIR_{WRP}$ set), otherwise *Reserved*
**Access Mode:** *Privileged*, accessible only when access to Coprocessor 0 is enabled

The MSA Save register (*MSASave*) is a 32-bit read/write register specifying which of the 32 architecturally defined vector registers W0, …, W31 have not been saved after a software context switch. Figure 3-18 shows the format of the *MSASave*. The reset state of the MSA Save register is zero.

The software can read and write the *MSASave* using CFCMSA and CTCMSA (Copy From and To Control MSA register) instructions. If the multi-threading module is present, each thread context has its own *MSASave* register instance.

If both bit $MSAAccess_{Wn}$ and bit $MSASave_{Wn}$ are set, where *n* = 0, …, 31, then register W*n* has to be saved on behalf of the previous software context and restored with the value corresponding to the current context.

**Figure 3-18 MSASave Register Format**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| W31 | W30 | W29 | W28 | W27 | W26 | W25 | W24 | W23 | W22 | W21 | W20 | W19 | W18 | W17 | W16 | W15 | W14 | W13 | W12 | W11 | W10 | W9 | W8 | W7 | W6 | W5 | W4 | W3 | W2 | W1 | W0 |

## 3.4.5 MSA Modify Register (MSAModify, MSA Control Register 4)

**Compliance Level:** *Required* for vector registers partitioning (i.e. $MSAIR_{WRP}$ set), otherwise *Reserved*
**Access Mode:** *Privileged*, accessible only when access to Coprocessor 0 is enabled

The MSA Modify register (*MSAModify*) is a 32-bit read/write register specifying which of the 32 architecturally defined vector registers W0, …, W31 have been modified (written). Figure 3-13 shows the format of the *MSAModify*. The reset state of the MSA Modify register is zero.

The software can read and write the *MSAModify* using CFCMSA and CTCMSA (Copy From and To Control MSA register) instructions. If the multi-threading module is present, each thread context has its own *MSAModify* register instance.

*MSAModify* is updated by the hardware when the execution of each MSA or FPU instruction completes. The update is a logical `or` operation, i.e. hardware updates never clear any bits in *MSAModify* register.

If bit $MSAModify_{Wn}$ is set, where $n = 0, \ldots 31$, then the software has been granted access to and has modified register W*n* since the last time the software cleared bit *n*.

**Figure 3-19 MSAModify Register Format**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| W31 | W30 | W29 | W28 | W27 | W26 | W25 | W24 | W23 | W22 | W21 | W20 | W19 | W18 | W17 | W16 | W15 | W14 | W13 | W12 | W11 | W10 | W9 | W8 | W7 | W6 | W5 | W4 | W3 | W2 | W1 | W0 |

## 3.4.6 MSA Request Register (MSARequest, MSA Control Register 5)

**Compliance Level:** *Required* for vector registers partitioning (i.e. $MSAIR_{WRP}$ set), otherwise *Reserved*
**Access Mode:** *Privileged*, accessible only when access to Coprocessor 0 is enabled

The MSA Request register (*MSARequest*) is a 32-bit read-only register specifying which of the 32 architecturally defined vector registers W0, …, W31 the current MSA or FPU instruction has requested access to but are not yet available, i.e. $MSAAcces_{Wn}$ is clear, or are not yet saved, i.e. $MSASave_{Wn}$ is set. Figure 3-13 shows the format of the *MSARequest*. The reset state of the MSA Request register is zero.

The software can read the *MSARequest* using CFCMSA (Copy From Control MSA register) instruction. If the multi-threading module is present, each thread context has its own *MSARequest* register instance.

*MSARequest* is set by the hardware for each MSA or FPU instruction with all vector registers the instruction will access in either read or write mode. *MSARequest* is always cleared before setting the bits for the current MSA or FPU instruction.

**Figure 3-20 MSARequest Register Format**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| W31 | W30 | W29 | W28 | W27 | W26 | W25 | W24 | W23 | W22 | W21 | W20 | W19 | W18 | W17 | W16 | W15 | W14 | W13 | W12 | W11 | W10 | W9 | W8 | W7 | W6 | W5 | W4 | W3 | W2 | W1 | W0 |

## 3.4.7 MSA Map Register (MSAMap, MSA Control Register 6)

**Compliance Level:** *Required* for vector registers partitioning (i.e. $MSAIR_{WRP}$ set), otherwise *Reserved*
**Access Mode:** *Privileged*, accessible only when access to Coprocessor 0 is enabled

The MSA Map register (*MSAMap*) is a 32-bit read/write register specifying a vector register to be mapped. Figure 3-21 shows the format of the *MSAMap*. Figure 3-22 describes the *MSAMap* fields.

The software can read and write the *MSAMap* using CFCMSA and CTCMSA (Copy From and To Control MSA register) instructions. If the multi-threading module is present, each thread context has its own *MSAMap* register instance.

When value *n*, *n* = 0, …, 31, is written to *MSAMap,* the hardware is instructed to map vector register W*n* to one of the available physical registers. The successful mapping is confirmed by setting $MSAAccess_{Wn}$.

The total number of vector registers mapped at any time can not exceed the number of physical registers implemented.

**Figure 3-21 MSAMap Register Format**

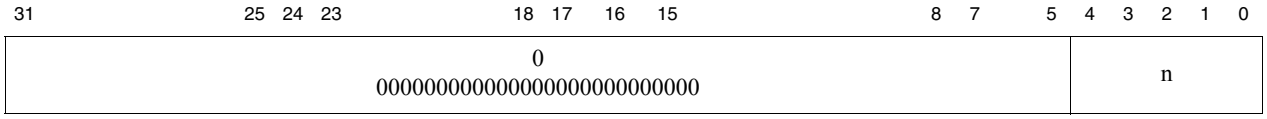



**Figure 3-22 MSAMap Register Field Descriptions**

| Fields | | Description | Read/ Write | Reset State | Compliance |
|---|---|---|---|---|---|
| Name | Bits | | | | |
| 0 | 31:5 | Reserved for future use; reads as zero and must be written as zero. | R0 | 0 | Reserved |
| n | 4:0 | Vector register index. | R/W | 0 | Required |

## 3.4.8 MSA Unmap Register (MSAUnmap, MSA Control Register 7)

**Compliance Level:** *Required* for vector registers partitioning (i.e. $MSAIR_{WRP}$ set), otherwise *Reserved*
**Access Mode:** *Privileged*, accessible only when access to Coprocessor 0 is enabled

The MSA Unmap register (*MSAUnmap*) is a 32-bit read/write register specifying a vector register to be unmapped. Figure 3-23 shows the format of the *MSAUnmap*. Figure 3-24 describes the *MSAUnmap* fields.

The software can read and write the *MSAUnmap* using CFCMSA and CTCMSA (Copy From and To Control MSA register) instructions. If the multi-threading module is present, each thread context has its own *MSAUnmap* register instance.

When value *n*, *n* = 0, …, 31, is written to *MSAUnmap,* the hardware is instructed to unmap vector register W*n*. The unmapping is confirmed by clearing $MSAAccess_{Wn}$.

**Figure 3-23 MSAUnmap Register Format**

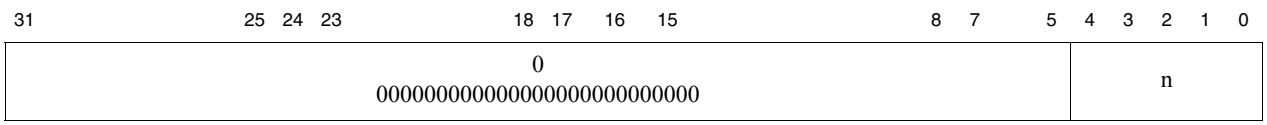


**Figure 3-24 MSAUnmap Register Field Descriptions**

| Fields | | Description | Read/ Write | Reset State | Compliance |
|---|---|---|---|---|---|
| Name | Bits | | | | |
| 0 | 31:5 | Reserved for future use; reads as zero and must be written as zero. | R0 | 0 | Reserved |
| n | 4:0 | Vector register index. | R/W | 0 | Required |

## 3.5 Exceptions

MSA instructions can generate the following exceptions (see Table 3.5):

- *Reserved Instruction*, if bit $Config3_{MSAP}$ (CP0 Register 16, Select 3, bit 28) is not set, or if the usable FPU operates in 32-bit mode, i.e. bit $Status_{CU1}$ (CP Register 12, Select 0, bit 29) is set and bit $Status_{FR}$ (CP Register 12, Select 0, bit 26) is not set. This exception uses the common exception vector with ExcCode field in *Cause* CP0 register set to 0x0a.
- *Coprocessor Unusable*, if CFCMSA or CTCMSA instructions attempt to read or write privileged MSA control registers without Coprocessor 0 access enabled. This exception uses the common exception vector with ExcCode field in *Cause* CP0 register set to 0x0b and CE field set to 0 to indicate Coprocessor 0.
- *MSA Disabled*, if bit $Config5_{MSAEn}$ (CP0 Register 16, Select 5, bit 27) is not set or, when vector registers partitioning is enabled (i.e. $MSAIR_{WRP}$ set), if any MSA vector register accessed by the instruction is either not available or needs to be saved/restored due to a software context switch. This exception uses the common exception vector with ExcCode field in *Cause* CP0 register set to 0x15.
- *MSA Floating Point*, a data dependent exception signaled by the MSA floating point instruction. This exception uses the common exception vector with ExcCode field in *Cause* CP0 register set to 0x0e. The exact reason for taking this exception is in the *Cause* bits of the MSA Control and Status Register *MSACSR*.

All MSA reserved opcodes in Table 3.18 are considered to be part of the MIPS SIMD Architecture on cores implementing MSA. These opcodes will generate the following exceptions (see Table 3.5):

- *MSA Disabled*, if MSA instructions are not enabled.
- *Reserved Instruction*, if MSA instructions are enabled.

The conditions under which the MSA instructions are enabled are documented in Section 3.2 "MSA Software Detection" and Section 3.3.2 "Floating-Point Registers Mapping".

**Table 3.5 MSA Exception Code (ExcCode) Values**

| Exception Code Value | | Mnemonic | Description |
|---|---|---|---|
| Decimal | Hexadecimal | | |
| 10 | 0x0a | RI | Reserved Instruction exception |
| 11 | 0x0b | CpU | Coprocessor Unusable exception |
| 14 | 0x0e | MSAFPE | MSA Floating Point exception |
| 21 | 0x15 | MSADis | MSA Disabled exception |

## 3.5.1 Handling the MSA Disabled Exception

The exact reason for taking a MSA Disabled Exception can be determined by checking the $Config5_{MSAEn}$ bit. No MSA instruction can be executed if this bit is not set. By setting $Config5_{MSAEn}$, the OS knows the current software context uses MSA resources and therefore it will save/restore MSA registers on context switch.

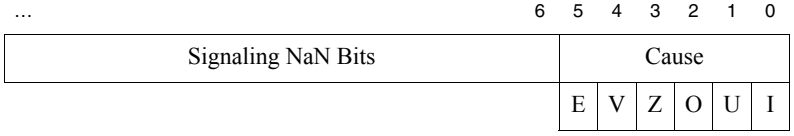
If the vector registers partitioning is implemented (i.e. $MSAIR_{WRP}$ is set), the MSA Disabled Exception could be signaled even if $Config5_{MSAEn}$ bit is set. In this instance, the exception is caused by some vector registers not being ready (either not available or in need to be saved/restored) for the current software context. The OS can map or save/restore these vector registers by examining *MSARequest*, *MSAAccess*, and *MSASave*.

See Appendix A, "Vector Registers Partitioning" for an example of handling the MSA Disabled Exception when vector registers partitioning is implemented.

## 3.5.2 Handling the MSA Floating Point Exception

In normal operation mode, floating point exceptions are signaled if at least one vector element causes an exception enabled by the *MSACSR* Enable bitfield. There is no precise indication in this case on which elements are at fault and the corresponding exception causes. The exception handling routine should set the *MSACSR* non-trapping exception mode bit NX and re-execute the MSA floating point instruction. All elements which would normally signal an exception according to the *MSACSR* Enable bitfield are set to signaling NaN values, where the least significant 6 bits have the same format as the *MSACSR* Cause field (see Figure 3-25, Table 3.3) to record the specific exception or exceptions detected for that element. The other elements will be set to the calculated results based on their operands.

**Figure 3-25 Output Format for Faulting Elements when NX is set**



When the non-trapping exception mode bit NX is set, no floating point exception will be taken, not even the always enabled Unimplemented Operation Exception. Note that by setting the NX bit, the *MSACSR* Enable bitfield is not changed and is still used to generate the appropriate default results. Regardless of the NX value, if a floating point exception is not enabled, i.e. the corresponding *MSACSR* Enable bit is 0, the floating point result is a default value as shown in Table 3.6.

The pseudocode in Figure 3.26 shows the process of updating the *MSACSR* Cause bits and setting the destination's value. This process is invoked element-by-element for all elements the instruction operates on. It is assumed *MSACSR* Cause bits are all cleared before executing the instruction. The *MSACSR* Flags bits are updated after all the elements have been processed and *MSACSR* Cause contains no enabled exceptions. If there are enabled exceptions in *MSACSR* Cause, a MSA floating-point exception will be signaled and the *MSACSR* Flags are not updated. The pseudocode in Figure 3.27 describes the *MSACSR* Flags update and exception signaling condition.

For instructions with non floating-point results, the pseudocode in Figure 3.26 and Figure 3.27 apply unchanged and both the format in Figure 3-25 and the default values from Table 3.6 are preserved for enabled exceptions when NX bit is set. For disabled exceptions, the default values are explicitly documented case-by-case in the instruction's description section.

**Table 3.6 Default Values for Floating Point Exceptions**

| Exception | Rounding Mode | Default Value, Disabled Exception | Default Value, Enabled Exception, and NX set |
|---|---|---|---|
| Invalid Operation | | The default value is either the default quiet NaN (see Table 3.7), or one of the signaling NaN operands propagated as a quiet NaN. | The default signaling NaN (see Table 3.7) of the format shown in Figure 3-25 with Cause V bit set. |
| Divide by Zero | | The default value is the properly signed infinity. | The default signaling NaN (see Table 3.7) of the format shown in Figure 3-25 with Cause Z bit set. |
| Underflow | | The default value is the rounded result based on the rounding mode. | The default signaling NaN (see Table 3.7) of the format shown in Figure 3-25 with Cause U bit set. |
| Inexact | | The default value is the rounded result based on the rounding mode. If caused by an overflow without the overflow exception enabled, the default value is the overflowed result. | The default signaling NaN (see Table 3.7) of the format shown in Figure 3-25 with Cause I bit set. |
| Overflow | | The default value depends on the rounding mode, as shown below. | The default signaling NaN (see Table 3.7) of the format shown in Figure 3-25 with Cause O bit set. |
| | Round to nearest | An infinity with the sign of the overflow value. | |
| | Round toward zero | The format's largest finite number with the sign of the overflow value. | |
| | Round towards positive | For positive overflow values, positive infinity. For negative overflow values, the format's smallest negative finite number. | |
| | Round towards negative | For positive overflow values, the format's largest finite number. For negative overflow values, minus infinity. | |

### Table 3.7 Default NaN Encodings

| Format | Quiet NaN | Signaling NaN |
|---|---|---|
| 16-bit | 0x7E00 | 0x7CNN[1] |
| 32-bit | 0x7FC0 0000 | 0x7F80 00NN |
| 64-bit | 0x7FF8 0000 0000 0000 | 0x7FF0 0000 0000 00NN |

1. All signaling NaN values have the format shown in Figure 3-25. Byte 0xNN has at least one bit set showing the reason for generating the signaling NaN value.

### Figure 3.26 $MSACSR_{Cause}$ Update Pseudocode

```
Input
   c: current element exception(s) E, V, Z, O, U, I bitfield
      (bit E is 0x20, O is 0x04, U is 0x02, and I is 0x01)
   d: default value to be used in case of a disabled exception
   e: signaling NaN value to be used in case of NX set, i.e. a non-trapping
      exception
   r: result value if the operation completed without an exception

Output
   v: value to be written to destination element
   Updated MSACSR_Cause

enable ← MSACSR_Enable | E /* Unimplemented (E) is always enabled */

/* Set Inexact (I) when Overflow (O) is not enabled (see Table 3.3) */
if (c & O) ≠ 0 and (enable & O) = 0 then
   c ← c | I
endif

/* Clear Exact Underflow when Underflow (U) is not enabled (see Table 3.3) */
if (c & U) ≠ 0 and (enable & U) = 0 and (c & I) = 0 then
   c ← c ^ U
endif

cause ← c & enable

if cause = 0 then
   /* No enabled exceptions, update the MSACSR Cause with all current exceptions */
   MSACSR_Cause ← MSACSR_Cause | c

   if c = 0 then
      /* Operation completed successfully, destination gets the result */
      v ← r
   else
      /* Current exceptions are not enabled, destination
         gets the default value for disabled exceptions case */
      v ← d
```

```
    endif
else
    /* Current exceptions are enabled */
    if MSACSR_NX = 0 then
        /* Exceptions will trap, update MSACSR Cause with all current exceptions,
           destination is not written */
        MSACSR_Cause ← MSACSR_Cause | c
    else
        /* No trap on exceptions, element not recorded in MSACSR Cause,
           destination gets the signaling NaN value for non-trapping exception */
        v ← ((e >> 6) << 6) | c
    endif
endif
```

**Figure 3.27 $MSACSR_{Flags}$ Update and Exception Signaling Pseudocode**

```
if (MSACSR_Cause & (MSACSR_Enable | E)) = 0 then /* Unimplemented (bit E 0x20)
                                                    is always enabled */
    /* No enabled exceptions, update the MSACSR Flags with all exceptions */
    MSACSR_Flags ← MSACSR_Flags | MSACSR_Cause
else
    /* Trap on the exceptions recorded in MSACSR Cause,
       MSACSR Flags are not updated */
    SignalException(MSAFPE, MSACSR_Cause)
```

### 3.5.3 NaN Propagation

MSA propagates NaN operands as specified by the IEEE Standard for Floating-Point Arithmetic 754™-2008.

If the destination format is floating-point, all NaN propagating operations with one NaN operand produce a NaN with the payload of the input NaN. When two or three operands are NaN, the payload of the resulting NaN is identical to the payload of one of the input NaNs selected from left to right as described by the instruction format.

The above NaN propagation rules apply to select the signaling NaN operand used in generating the default quiet NaN value when the Invalid Operation exception is disabled (see Table 3.6).

Note that signaling NaN operands always signal the Invalid Operation exception and as such, they take precedence over all quiet NaN operands.

If the destination format is not floating-point (e.g. conversions to integer/fixed-point or compares) or the NaN operands are not propagated (e.g. min or max operations), the expected result is documented in the instruction's description section.

Quiet NaN values are generated from input signaling NaN values by:

- Copying the signaling NaN sign value to the quiet NaN sign
- Copying the most significant bits of the signaling NaN mantissa to the most significant bits of the quiet NaN mantissa. In cases where the source signaling NaN and destination quiet NaN have the same width, all mantissa

bits are copied. In cases where the destination is wider than the source, the least significant bits of the destination mantissa are set to zero. In cases where the destination is narrower than the source, the least significant bits of the input mantissa are ignored.

- Setting the quiet NaN's exponent field to the maximum value and the most significant mantissa bit to 1.

### 3.5.4 Flush to Zero and Exception Signaling

Some MSA floating point instructions might not handle subnormal input operands or compute tiny non-zero results. Such instructions may signal the Unimplemented Operation Exception and let the software emulation finalize the operation. If software emulation is not needed or desired, *MSACSR* FS bit could be set to replace every tiny non-zero result and subnormal input operand with zero of the same sign.

The *MSACSR* FS bit changes the behavior of the Unimplemented Operation Exception. All the other floating point exceptions are signaled according to the new values of the operands or the results. In addition, when *MSACSR* FS bit is set:

- Tiny non-zero results are detected before rounding[1]. Flushing of tiny non-zero results causes Inexact and Underflow Exceptions to be signaled for all instructions except the approximate reciprocals.
- Flushing of subnormal input operands in all instructions except comparisons causes Inexact Exception to be signaled.
- For floating-point comparisons, the Inexact Exception is not signaled when subnormal input operands are flushed.
- 16-bit floating-point values and inputs to non arithmetic floating-point instructions are never flushed.

Should the alternate exception handling attributes of the IEEE Standard for Floating-Point Arithmetic 754™-2008, Section 8 be desired, the *MSACSR* FS bit should be zero, the Underflow Exception be enabled and a trap handler be provided to carry out the execution of the alternate exception handling attributes.

## 3.6 Instruction Syntax

The MSA assembly language coding uses the following syntax elements:

- *func*: function/instruction name, e.g. ADDS_S or adds_s for signed saturated add
- *df*: destination data format, which could be a byte, halfword, word, doubleword, or the vector itself
- *wd*, *ws*, and *wt*: destination, source, and target vector registers, e.g. $w0, …, $w31
- *rd*, *rs*: general purpose registers (GPRs), e.g. $0, …, $31
- *ws[n]*: vector register element of index *n*, where *n* is a valid index value for elements of data format *df*
- *m*: immediate value valid as a bit index for the data format *df*

[1] Tiny non-zero results that would have been normal after rounding are flushed to zero.

- *uN*, *sN*: *N*-bit unsigned or signed value, e.g. *s10*, *u5*
- *iN*: *N*-bit value where the sign is not relevant, e.g. *i8*

MSA instructions have two or three register, immediate, or element operands. One of the destination data format abbreviations shown in Table 3.8 is appended to the instruction name[2]. Note that the data format abbreviation is the same regardless of the instruction's assumed data type. For example all integer, fixed-point, and floating-point instructions operating on 32-bit elements use the same word (".W" in Table 3.8) data format.

**Table 3.8 Data Format Abbreviations**

| Data Format | Abbreviation |
|---|---|
| Byte, 8-bit | .B |
| Halfword16-bit | .H |
| Word, 32-bit | .W |
| Doubleword, 64-bit | .D |
| Vector | .V |

## 3.6.1 Vector Element Selection

MSA instructions of the form *func.df wd,ws[n]* and *func.df rd,ws[n]* select the $n^{th}$ element in the vector register *ws* based on the data format *df*. The valid element index values for various data formats and vector register sizes are shown in Table 3.9. The vector element is being used as a fixed operand across all destination vector elements.

**Table 3.9 Valid Element Index Values**

| Data Format | Element Index |
|---|---|
| Byte | $n = 0, \ldots, 15$ |
| Halfword | $n = 0, \ldots, 7$ |
| Word | $n = 0, \ldots, 3$ |
| Doubleword | $n = 0, 1$ |

## 3.6.2 Load/Store Offsets

The vector load and store instructions take a 10-bit signed offset *s10* in data format *df* units. By convention, in the assembly language syntax all offsets are in bytes and have to be multiple of the size of the data format.

[2] Instructions names and data format abbreviations are case insensitive.

For example, the offset indicated by the load word vector instruction

```
ld.w $w5,12($1)
```

is not 12 words, but rather 12 bytes. The assembler divides the byte offset (i.e. 12) by the size of the word data format (i.e. 4), and generates the LD.W machine instruction by setting *s10* bitfield to the word offset value (i.e. 3 = 12 / 4).

### 3.6.3 Instruction Examples

Let us assume vector registers $w1 and $w2 are initialized to the word values shown in Figure 3-28, Figure 3-29 and GPR $2 is initialized as shown in Figure 3-30.

**Figure 3-28 Source Vector $w1 Values**

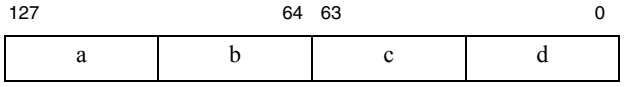



**Figure 3-29 Source Vector $w2 Values**

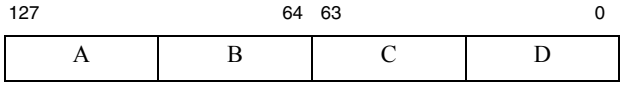
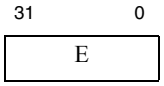



**Figure 3-30 Source GPR $2 Value**




Regular MSA instructions operate element-by-element with identical source, target, and destination data types. Figure 3-31 through Figure 3-34 have the resulting values of destination vectors $w4, $w5, $w6, and $w7 after executing the following sequence of word additions and move instructions:

```
addv.w $w5,$w1,$w2
fill.w $w6,$2
addvi.w $w7,$w1,17
splati.w $w8,$w2[2]
```

**Figure 3-31 Destination Vector $w5 Value for ADDV.W Instruction**

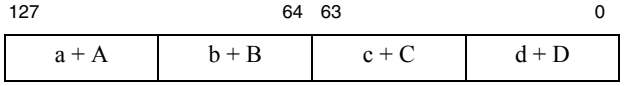


**Figure 3-32 Destination Vector $w6 Value for FILL.W Instruction**

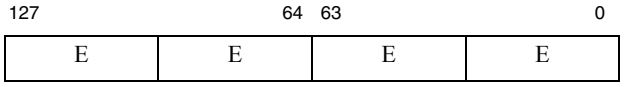



**Figure 3-33 Destination Vector $w7 Value for ADDVI.W Instruction**

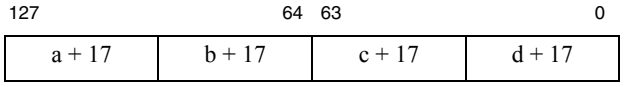
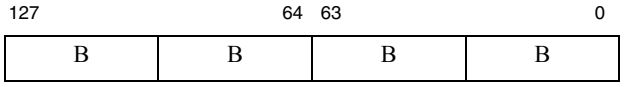



**Figure 3-34 Destination Vector $w8 Value for SPLAT.W Instruction**

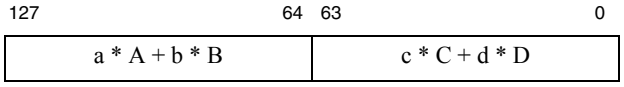



Other MSA instructions operate on adjacent odd/even source elements generating results on data formats twice as wide. See Figure 3-35 for the destination layout of such an instruction, i.e. the signed doubleword dot product:

```
dotp_s.d $w9,$w1,$w2
```

Note that the actual instruction, e.g. DOTP_S.D, specifies the data format of the destination. The data format of the source operands is inferred as being also signed and half the width, i.e. word in this case.

**Figure 3-35 Destination Vector $w9 Value for DOTP_S Instruction**




# 3.7 Instruction Encoding

## 3.7.1 Data Format and Index Encoding

Most of the MSA instructions operate on byte, halfword, word or doubleword data formats (see Section 3.3 "MSA Vector Registers"). Internally, the data format *df* is coded by a 2-bit field as shown in Table 3.10. For instructions operating only on two data formats, the internal coding is shown in Table 3.11 and Table 3.12.

### Table 3.10 Two-bit Data Format Field Encoding

| df | *Bit 0* | |
|---|---|---|
| *Bit 1* | 0 | 1 |
| 0 | Byte | Halfword |
| 1 | Word | Doubleword |

### Table 3.11 Halfword/Word Data Format Field Encoding

| df | *Bit 0* | |
|---|---|---|
| | 0 | 1 |
| | Halfword | Word |

### Table 3.12 Word/Doubleword Data Format Field Encoding

| df | *Bit 0* | |
|---|---|---|
| | 0 | 1 |
| | Word | Doubleword |

### Table 3.13 Data Format and Element Index Field Encoding

| df/n[1] | *Bits 5...0* | | | |
|---|---|---|---|---|
| | 00nnnn | 100nnn | 1100nn | 11100n |
| | Byte | Halfword | Word | Doubleword |

| df/n | *Bits 5...0* | | | |
|---|---|---|---|---|
| | 01nnnn | 101nnn | 1101nn | 11101n |
| | Reserved | | | |

1. Bits marked as *n* give the element index value.

### Table 3.14 Data Format and Bit Index Field Encoding

| df/m[1] | *Bits 6...0* | | | |
|---|---|---|---|---|
| | 0mmmmmm | 10mmmmm | 110mmmm | 1110mmm |
| | Doubleword | Word | Halfword | Byte |

1. Bits marked as *m* give the bit index value.

MSA instructions using a specific vector element code both data format and element index in a 6-bit field *df/n* as shown in Table 3.13. All invalid index values or data formats will generate a Reserved Instruction Exception. For example, a vector register has 16 byte elements while the byte data format can code up to 32 byte elements. Selecting any vector byte element other than 0, 1, …, 15 generates a Reserved Instruction Exception.

The combinations marked Vector (".V" in Table 3.8) are used for coding certain instructions with data formats other than byte, halfword, word, or doubleword.

If an instruction specifies a bit position, the data format and bit index *df/m* are coded as shown in Table 3.14.

### 3.7.2 Instruction Formats

All MSA instructions except branches use 40 minor opcodes in the MSA major opcode 30 (see Table 3.16). MSA branch instructions use 10 *rs* field encodings in the COP1 opcode 17 (see Table 3.17).

Each allocated minor opcode is associated specific instruction formats as follows:

- I8 (Figure 3-36): instructions with an 8-bit immediate value and either implicit data format or data format df (Table 3.8) coded in bits 25…24
- I5 (Figure 3-37): instructions with a 5-bit immediate value, where the data format df (Table 3.8) is coded in bits 22…21 and the operation in bits 25…23
- BIT (Figure 3-38): instructions with an immediate bit index and data format df/m (Table 3.14) coded in bits 22…16, where the operation is coded in bits 25…23
- I10 (Figure 3-39): instructions with a 10-bit immediate, where the data format df (Table 3.8) is coded in bits 22…21 and the operation in bits 25…23
- 3R (Figure 3-40): 3-register operations coded in bits 25…23 with data format df (Table 3.8) is coded in bits 22…21
- ELM (Figure 3-41): instructions with an immediate element index and data format df/n (Table 3.13) coded in bits 21…16, where the operation is coded in bits 25…22
- 3RF (Figure 3-42): 3-register floating-point or fixed-point operations coded in bits 25…22 with data format df (Table 3.11, Table 3.12) coded in bit 21
- VEC (Figure 3-43): 3-register instructions with implicit data formats depending on the operations coded in bits 25…21
- MI10 (Figure 3-44): 2-register instructions with a 10-bit immediate value, where the data format is either implicit or explicitly coded as df (Table 3.8) in bits 1…0, and the operation is coded in bit 25 and the minor opcode bits 5…2
- 2R (Figure 3-45): 2-register operations coded in bits 25…18 with data format df (Table 3.11) is coded in bits 17…16
- 2RF (Figure 3-46): 2-register floating-point operations coded in bits 25…17 with data format df (Table 3.11) coded in bit 16
- Branch (Figure 3-47): instructions with a 16-bit immediate, where the data format is either implicit or explicitly coded as df (Table 3.8) in bits 22…21, and the operation is coded in bits 25…23

**Figure 3-36 I8 Instruction Format**

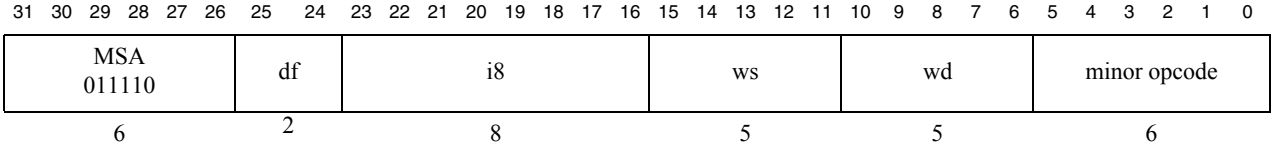



**Figure 3-37 I5 Instruction Format**

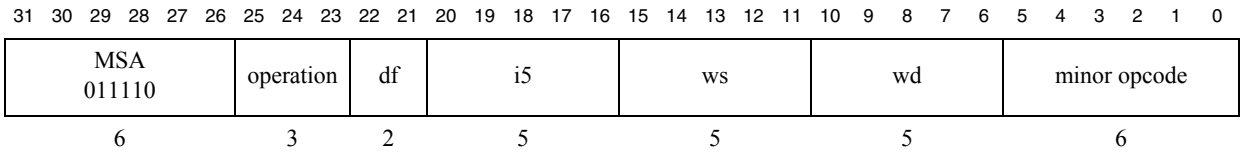



**Figure 3-38 BIT Instruction Format**

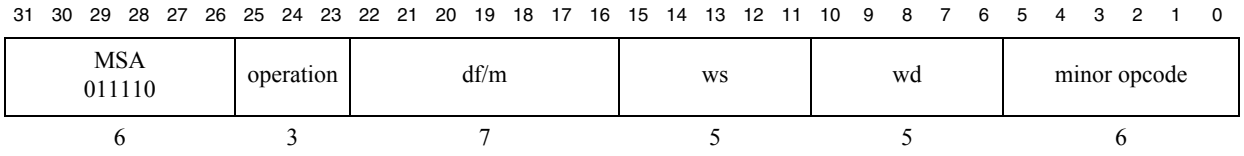



**Figure 3-39 I10 Instruction Format**

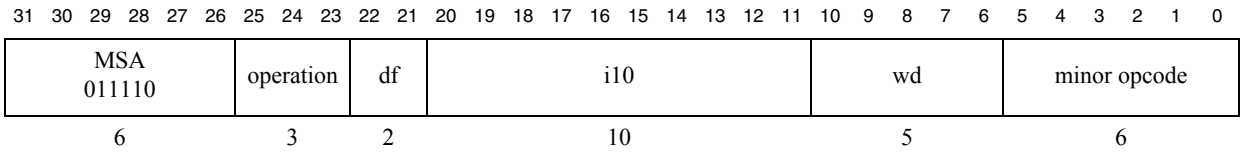



**Figure 3-40 3R Instruction Format**

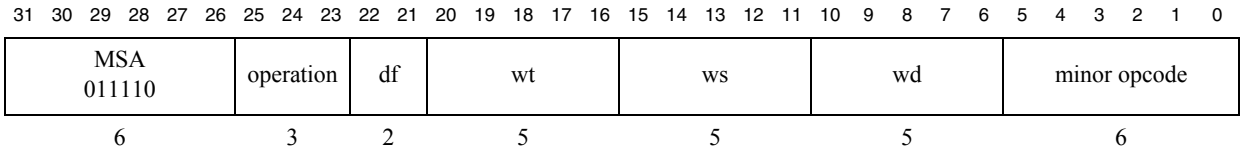


**Figure 3-41 ELM Instruction Format**

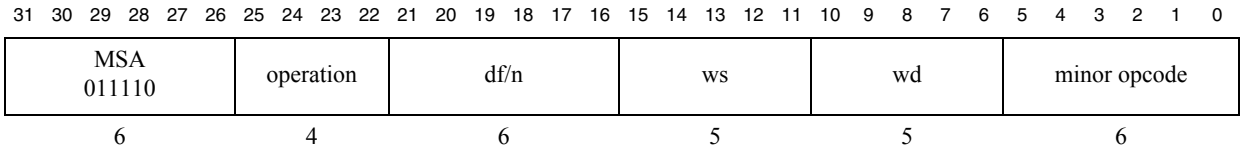



**Figure 3-42 3RF Instruction Format**

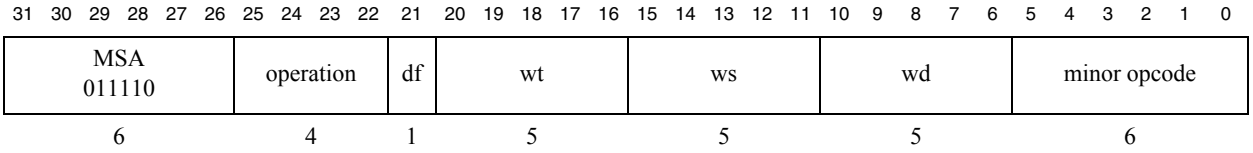



**Figure 3-43 VEC Instruction Format**

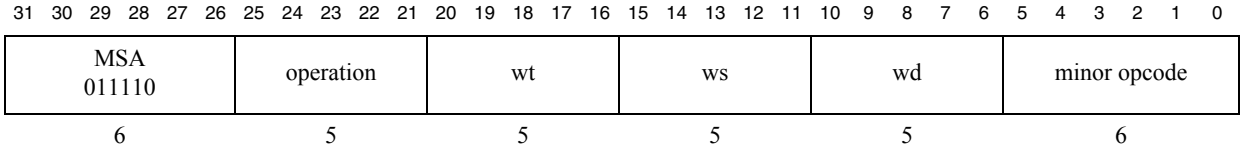



**Figure 3-44 MI10 Instruction Format**

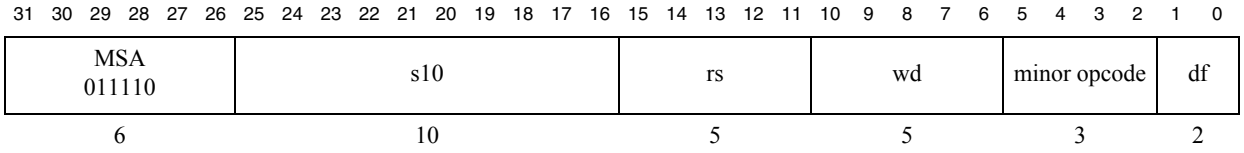



**Figure 3-45 2R Instruction Format**

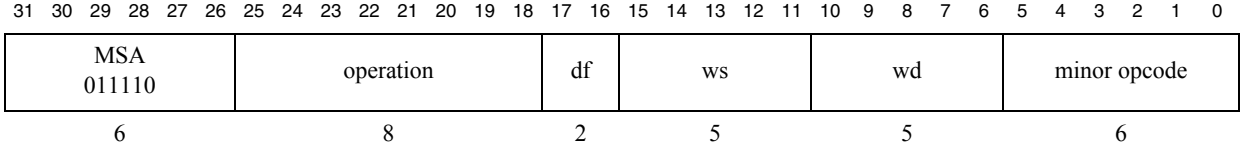


**Figure 3-46 2RF Instruction Format**

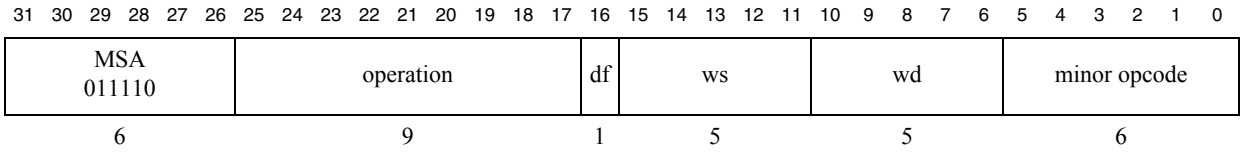
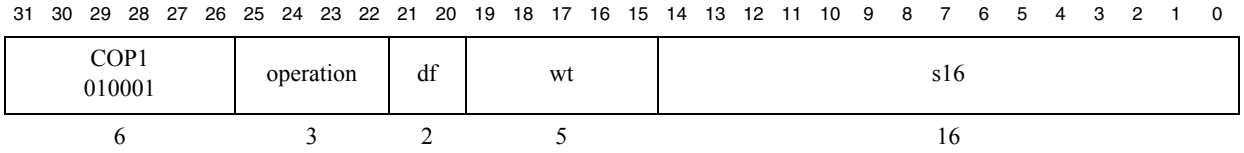



**Figure 3-47 Branch Instruction Format**




## 3.7.3 Instruction Bit Encoding

This chapter describes the bit encoding tables used for the MSA. Table 3.15 describes the meaning of the symbols used in the tables. These tables only list the instruction encoding for the MSA instructions. See Volumes I and II of this multi-volume set for a full encoding of all instructions.

Figure 3.48 shows a sample encoding table and the instruction *opcode* field this table encodes. Bits 31…29 of the *opcode* field are listed in the left-most columns of the table. Bits 28…26 of the *opcode* field are listed along the top-most rows of the table. Both decimal and binary values are given, with the first three bits designating the row, and the last three bits designating the column.

An instruction's encoding is found at the intersection of a row (bits 31…29) and column (bits 28…26) value. For instance, the *opcode* value for the instruction labelled EX1 is 33 (decimal, row and column), or 011011 (binary). Similarly, the *opcode* value for EX2 is 64 (decimal), or 110100 (binary).

**Figure 3.48 Sample Bit Encoding Table**

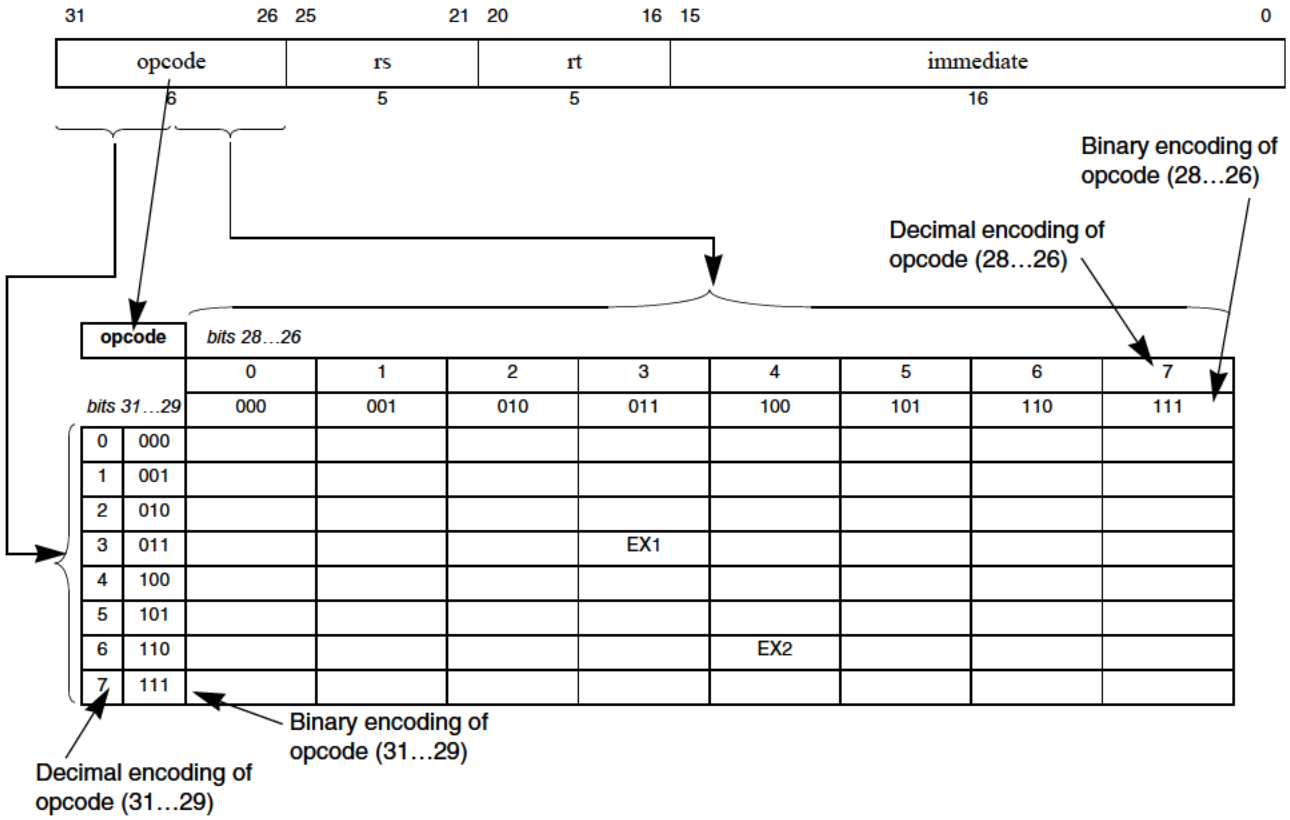



**Table 3.15 Symbols Used in the Instruction Encoding Tables**

| Symbol | Meaning |
|---|---|
| * | Operation or field codes marked with this symbol are reserved for future use. Executing such an instruction must cause a Reserved Instruction Exception. |
| δ | (Also *italic* field name.) Operation or field codes marked with this symbol denotes a field class. The instruction word must be further decoded by examining additional tables that show values for another instruction field. |
| β | Operation or field codes marked with this symbol represent a valid encoding for a higher-order MIPS ISA level. Executing such an instruction must cause a Reserved Instruction Exception. |
| ⊥ | Operation or field codes marked with this symbol represent instructions which are not legal if the processor is configured to be backward compatible with MIPS32 processors. If the processor is executing in Kernel Mode, Debug Mode, or 64-bit instructions are enabled, execution proceeds normally. In other cases, executing such an instruction must cause a Reserved Instruction Exception (non-coprocessor encoding or coprocessor instruction encoding for a coprocessor to which access is allowed) or a Coprocessor Unusable Exception (coprocessor instruction encoding for a coprocessor to which access is not allowed). |

### Table 3.15 Symbols Used in the Instruction Encoding Tables

| Symbol | Meaning |
|---|---|
| θ | Operation or field codes marked with this symbol are available to licensed MIPS partners. To avoid multiple conflicting instruction definitions, MIPS Technologies will assist the partner in selecting appropriate encoding if requested by the partner. The partner is not required to consult with MIPS Technologies when one of these encoding is used. If no instruction is encoded with this value, executing such an instruction must cause a Reserved Instruction Exception (*SPECIAL2* encoding or coprocessor instruction encoding for a coprocessor to which access is allowed) or a Coprocessor Unusable Exception (coprocessor instruction encoding for a coprocessor to which access is not allowed). |
| σ | Field codes marked with this symbol represent an EJTAG support instruction and implementation of this encoding is optional for each implementation. If the encoding is not implemented, executing such an instruction must cause a Reserved Instruction Exception. If the encoding is implemented, it must match the instruction encoding as shown in the table. |
| ε | Operation or field codes marked with this symbol are reserved for MIPS Application Specific Extensions. If the ASE is not implemented, executing such an instruction must cause a Reserved Instruction Exception. |
| ϕ | Operation or field codes marked with this symbol are obsolete and will be removed from a future revision of the MIPS64 ISA. Software should avoid using these operation or field codes. |
| ⊕ | Operation or field codes marked with this symbol are valid for Release 2 implementations of the architecture. Executing such an instruction in a Release 1 implementation must cause a Reserved Instruction Exception. |

### Table 3.16 MIPS64 Encoding of the Opcode Field

| **opcode** | | *bits 28…26* | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| *bits 31…29* | | 000 | 001 | 010 | 011 | 100 | 101 | 110 | 111 |
| 0 | 000 | | | | | | | | |
| 1 | 001 | | | | | | | | |
| 2 | 010 | | *COP1* δ | | | | | | |
| 3 | 011 | | | | | | | *MSA* εδ | |
| 4 | 100 | | | | | | | | |
| 5 | 101 | | | | | | | | |
| 6 | 110 | | | | | | | | |
| 7 | 111 | | | | | | | | |

### Table 3.17 MIPS64 *COP1* Encoding of *rs* Field for MSA Branch Instructions

| **rs** | | *bits 23…21* | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| *bits 25…24* | | 000 | 001 | 010 | 011 | 100 | 101 | 110 | 111 |
| 0 | 00 | | | | | | | | |
| 1 | 01 | | | | BZ.V | | | | BNZ.V |
| 2 | 10 | | | | | | | | |
| 3 | 11 | BZ.B | BZ.H | BZ.W | BZ.D | BNZ.B | BNZ.H | BNZ.W | BNZ.D |

## Table 3.18 Encoding of MIPS *MSA* Minor Opcode Field[1]

| minor | | Bits 2...0 | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| Bits 5...3 | | 000 | 001 | 010 | 011 | 100 | 101 | 110 | 111 |
| 0 | 000 | I8 | I8 | I8 | * | * | * | I5 | I5[2] |
| 1 | 001 | * | BIT | BIT | * | * | 3R | 3R | 3R |
| 2 | 010 | 3R | 3R | 3R | 3R | 3R | 3R | * | * |
| 3 | 011 | * | ELM | 3RF | 3RF | 3RF | * | VEC/2R/2RF | * |
| 4 | 100 | MI10 | MI10 | MI10 | MI10 | MI10 | MI10 | MI10 | MI10 |
| 5 | 101 | * | * | * | * | * | * | * | * |
| 6 | 110 | * | * | * | * | * | * | * | * |
| 7 | 111 | * | * | * | * | * | * | * | * |

1. The opcodes marked '*' are MSA reserved opcodes and will generate the *MSA Disabled* exception or the *Reserved Instruction* exception as specified in Section 3.5 "Exceptions".
2. Includes I10

## Table 3.19 Encoding of Operation Field for MI10 Instruction Formats

| operation | | | data format[1] | |
|---|---|---|---|---|
| Bits 5...2 | | | Bits 1...0 | |
| *8* | *1000* | LD | 00 | .B |
| | | | 01 | .H |
| | | | 10 | .W |
| | | | 11 | .D |
| *9* | *1001* | ST | 00 | .B |
| | | | 01 | .H |
| | | | 10 | .W |
| | | | 11 | .D |

1. See Table 3.8.

## Table 3.20 Encoding of Operation Field for I5 Instruction Format

| operation | | Bits 5…0 | | data format[1] | |
|---|---|---|---|---|---|
| | | 6 | 7 | | |
| Bits 25…23 | | 000110 | 000111 | Bits 22…21 | |
| 0 | 000 | ADDVI | CEQI | 00 | .B |
| | | | | 01 | .H |
| | | | | 10 | .W |
| | | | | 11 | .D |
| 1 | 001 | SUBVI | * | 00 | .B |
| | | | | 01 | .H |
| | | | | 10 | .W |
| | | | | 11 | .D |
| 2 | 010 | MAXI_S | CLTI_S | 00 | .B |
| | | | | 01 | .H |
| | | | | 10 | .W |
| | | | | 11 | .D |
| 3 | 011 | MAXI_U | CLTI_U | 00 | .B |
| | | | | 01 | .H |
| | | | | 10 | .W |
| | | | | 11 | .D |
| 4 | 100 | MINI_S | CLEI_S | 00 | .B |
| | | | | 01 | .H |
| | | | | 10 | .W |
| | | | | 11 | .D |
| 5 | 101 | MINI_U | CLEI_U | 00 | .B |
| | | | | 01 | .H |
| | | | | 10 | .W |
| | | | | 11 | .D |
| 6 | 110 | * | LDI[2] | 00 | .B |
| | | | | 01 | .H |
| | | | | 10 | .W |
| | | | | 11 | .D |
| 7 | 111 | * | * | 00 | .B |
| | | | | 01 | .H |
| | | | | 10 | .W |
| | | | | 11 | .D |

1. See Table 3.8.
2. I10 instruction format.

**Table 3.21 Encoding of Operation Field for I8 Instruction Format**

| operation | | Bits 5…0 | | |
|---|---|---|---|---|
| | | 0 | 1 | 2 |
| Bits 25…24 | | 000000 | 000001 | 000010 |
| 0 | 00 | ANDI.B | BMNZI.B | SHF.B |
| 1 | 01 | ORI.B | BMZI.B | SHF.H |
| 2 | 10 | NORI.B | BSELI.B | SHF.W |
| 3 | 11 | XORI.B | * | * |

**Table 3.22 Encoding of Operation Field for VEC/2R/2RF Instruction Formats**

| operation | | Bits 22…21 | | | |
|---|---|---|---|---|---|
| | | 0 | 1 | 2 | 3 |
| Bits 25…23 | | 00 | 01 | 10 | 11 |
| 0 | 000 | AND.V | OR.V | NOR.V | XOR.V |
| 1 | 001 | BMNZ.V | BMZ.V | BSEL.V | * |
| 2 | 010 | * | * | * | * |
| 3 | 011 | * | * | * | * |
| 4 | 100 | * | * | * | * |
| 5 | 101 | * | * | * | * |
| 6 | 110 | 2R format | 2RF format | * | * |
| 7 | 111 | * | * | * | * |

**Table 3.23 Encoding of Operation Field for 2R Instruction Formats**

| operation | | | data format[1] | |
|---|---|---|---|---|
| Bits 20…18 | | | Bits 17…16 | |
| 0 | 000 | FILL | 00 | .B |
| | | | 01 | .H |
| | | | 10 | .W |
| | | | 11 | .D |
| 1 | 001 | PCNT | 00 | .B |
| | | | 01 | .H |
| | | | 10 | .W |
| | | | 11 | .D |

**Table 3.23 Encoding of Operation Field for 2R Instruction Formats (Continued)**

| | | | | |
|---|---|---|---|---|
| 2 | 010 | NLOC | 00 | .B |
| | | | 01 | .H |
| | | | 10 | .W |
| | | | 11 | .D |
| 3 | 011 | NLZC | 00 | .B |
| | | | 01 | .H |
| | | | 10 | .W |
| | | | 11 | .D |
| 4…7 | 100…111 | * | 00 | .B |
| | | | 01 | .H |
| | | | 10 | .W |
| | | | 11 | .D |

1. See Table 3.8.

**Table 3.24 Encoding of Operation Field for 2RF Instruction Formats**

| operation | | | data format[1] | |
|---|---|---|---|---|
| *Bits 20…17* | | | *Bit 16* | |
| 0 | 0000 | FCLASS | 0 | .W |
| | | | 1 | .D |
| 1 | 0001 | FTRUNC_S | 0 | .W |
| | | | 1 | .D |
| 2 | 0010 | FTRUNC_U | 0 | .W |
| | | | 1 | .D |
| 3 | 0011 | FSQRT | 0 | .W |
| | | | 1 | .D |
| 4 | 0100 | FRSQRT | 0 | .W |
| | | | 1 | .D |
| 5 | 0101 | FRCP | 0 | .W |
| | | | 1 | .D |
| 6 | 0110 | FRINT | 0 | .W |
| | | | 1 | .D |
| 7 | 0111 | FLOG2 | 0 | .W |
| | | | 1 | .D |
| 8 | 1000 | FEXUPL | 0 | .W |
| | | | 1 | .D |
| 9 | 1001 | FEXUPR | 0 | .W |
| | | | 1 | .D |

**Table 3.24 Encoding of Operation Field for 2RF Instruction Formats (Continued)**

| | | | | |
|---|---|---|---|---|
| 10 | 1010 | FFQL | 0 | .W |
| | | | 1 | .D |
| 11 | 1011 | FFQR | 0 | .W |
| | | | 1 | .D |
| 12 | 1100 | FTINT_S | 0 | .W |
| | | | 1 | .D |
| 13 | 1101 | FTINT_U | 0 | .W |
| | | | 1 | .D |
| 14 | 1110 | FFINT_S | 0 | .W |
| | | | 1 | .D |
| 15 | 1111 | FFINT_U | 0 | .W |
| | | | 1 | .D |

1. See Table 3.12.

**Table 3.25 Encoding of Operation Field for 3R Instruction Format**

| operation | | *Bits 5…0* | | | | | | | | | data format[1] | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| *Bits 25…23* | | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | *Bits 22…21* | |
| | | 001101 | 001110 | 001111 | 010000 | 010001 | 010010 | 010011 | 010100 | 010101 | | |
| 0 | 000 | SLL | ADDV | CEQ | ADD_A | SUBS_S | MULV | * | SLD | VSHF | 00 | .B |
| | | | | | | | | DOTP_S | | | 01 | .H |
| | | | | | | | | | | | 10 | .W |
| | | | | | | | | | | | 11 | .D |
| 1 | 001 | SRA | SUBV | * | ADDS_A | SUBS_U | MADDV | * | SPLAT | SRAR | 00 | .B |
| | | | | | | | | DOTP_U | | | 01 | .H |
| | | | | | | | | | | | 10 | .W |
| | | | | | | | | | | | 11 | .D |
| 2 | 010 | SRL | MAX_S | CLT_S | ADDS_S | SUBSUS_U | MSUBV | * | PCKEV | SRLR | 00 | .B |
| | | | | | | | | DPADD_S | | | 01 | .H |
| | | | | | | | | | | | 10 | .W |
| | | | | | | | | | | | 11 | .D |
| 3 | 011 | BCLR | MAX_U | CLT_U | ADDS_U | SUBSUU_S | * | * | PCKOD | * | 00 | .B |
| | | | | | | | | DPADD_U | | | 01 | .H |
| | | | | | | | | | | | 10 | .W |
| | | | | | | | | | | | 11 | .D |
| 4 | 100 | BSET | MIN_S | CLE_S | AVE_S | ASUB_S | DIV_S | * | ILVL | * | 00 | .B |
| | | | | | | | | DPSUB_S | | HADD_S | 01 | .H |
| | | | | | | | | | | | 10 | .W |
| | | | | | | | | | | | 11 | .D |

**Table 3.25 Encoding of Operation Field for 3R Instruction Format (Continued)**

| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 5 | 101 | BNEG | MIN_U | CLE_U | AVE_U | ASUB_U | DIV_U | * | ILVR | * | 00 | .B |
| | | | | | | | | DPSUB_U | | HADD_U | 01 | .H |
| | | | | | | | | | | | 10 | .W |
| | | | | | | | | | | | 11 | .D |
| 6 | 110 | BINSL | MAX_A | * | AVER_S | * | MOD_S | * | ILVEV | * | 00 | .B |
| | | | | | | | | | | HSUB_S | 01 | .H |
| | | | | | | | | | | | 10 | .W |
| | | | | | | | | | | | 11 | .D |
| 7 | 111 | BINSR | MIN_A | * | AVER_U | * | MOD_U | * | ILVOD | * | 00 | .B |
| | | | | | | | | | | HSUB_U | 01 | .H |
| | | | | | | | | | | | 10 | .W |
| | | | | | | | | | | | 11 | .D |

1. See Table 3.8.

**Table 3.26 Encoding of Operation Field for ELM Instruction Format**

| operation | | | data format[1] | |
|---|---|---|---|---|
| *Bits 25…22* | | | *Bits 21…16* | |
| 0 | 0000 | SLDI | 00nnnn | .B |
| | | | 100nnn | .H |
| | | | 1100nn | .W |
| | | | 11100n | .D |
| | | * | 11110n | |
| | | CTCMSA | 111110 | |
| | | * | 111111 | |
| 1 | 0001 | SPLATI | 00nnnn | .B |
| | | | 100nnn | .H |
| | | | 1100nn | .W |
| | | | 11100n | .D |
| | | * | 11110n | |
| | | CFCMSA | 111110 | |
| | | * | 111111 | |
| 2 | 0010 | COPY_S | 00nnnn | .B |
| | | | 100nnn | .H |
| | | | 1100nn | .W |
| | | | 11100n | .D |
| | | * | 11110n | |
| | | MOVE.V | 111110 | |
| | | * | 111111 | |
| 3 | 0011 | COPY_U | 00nnnn | .B |
| | | | 100nnn | .H |
| | | | 1100nn | .W |
| | | | 11100n | .D |
| | | * | 11110n | |
| | | | 111110 | |
| | | | 111111 | |

**Table 3.26 Encoding of Operation Field for ELM Instruction Format (Continued)**

| | | | | |
|---|---|---|---|---|
| 4 | 0100 | INSERT | 00nnnn | .B |
| | | | 100nnn | .H |
| | | | 1100nn | .W |
| | | | 11100n | .D |
| | | * | 11110n | |
| | | | 111110 | |
| | | | 111111 | |
| 5 | 0101 | INSVE | 00nnnn | .B |
| | | | 100nnn | .H |
| | | | 1100nn | .W |
| | | | 11100n | .D |
| | | * | 11110n | |
| | | | 111110 | |
| | | | 111111 | |
| 6...15 | 0110...1111 | * | 00nnnn | |
| | | | 100nnn | |
| | | | 1100nn | |
| | | | 11100n | |
| | | * | 11110n | |
| | | | 111110 | |
| | | | 111111 | |

1. See Table 3.13.

**Table 3.27 Encoding of Operation Field for 3RF Instruction Format**

| **operation** | | *Bits 5...0* | | | | | | **data format**[1] |
|---|---|---|---|---|---|---|---|---|
| | | 26 | | 27 | | 28 | | *Bit 21* |
| *Bits 25...22* | | 011010 | | 011011 | | 011100 | | |
| 0 | 0000 | FCAF | .W | FADD | .W | * | .W | 0 |
| | | | .D | | .D | | .D | 1 |
| 1 | 0001 | FCUN | .W | FSUB | .W | FCOR | .W | 0 |
| | | | .D | | .D | | .D | 1 |
| 2 | 0010 | FCEQ | .W | FMUL | .W | FCUNE | .W | 0 |
| | | | .D | | .D | | .D | 1 |
| 3 | 0011 | FCUEQ | .W | FDIV | .W | FCNE | .W | 0 |
| | | | .D | | .D | | .D | 1 |
| 4 | 0100 | FCLT | .W | FMADD | .W | MUL_Q | .H | 0 |
| | | | .D | | .D | | .W | 1 |
| 5 | 0101 | FCULT | .W | FMSUB | .W | MADD_Q | .H | 0 |
| | | | .D | | .D | | .W | 1 |
| 6 | 0110 | FCLE | .W | * | | MSUB_Q | .H | 0 |
| | | | .D | | | | .W | 1 |
| 7 | 0111 | FCULE | .W | FEXP2 | .W | * | | 0 |
| | | | .D | | .D | | | 1 |
| 8 | 1000 | FSAF | .W | FEXDO | .H | * | .W | 0 |
| | | | .D | | .W | | .D | 1 |

**Table 3.27 Encoding of Operation Field for 3RF Instruction Format (Continued)**

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 9 | 1001 | FSUN | .W | * | | FSOR | .W | 0 |
| | | | .D | | | | .D | 1 |
| 10 | 1010 | FSEQ | .W | FTQ | .H | FSUNE | .W | 0 |
| | | | .D | | .W | | .D | 1 |
| 11 | 1011 | FSUEQ | .W | * | | FSNE | .W | 0 |
| | | | .D | | | | .D | 1 |
| 12 | 1100 | FSLT | .W | FMIN | .W | MULR_Q | .H | 0 |
| | | | .D | | .D | | .W | 1 |
| 13 | 1101 | FSULT | .W | FMIN_A | .W | MADDR_Q. | .H | 0 |
| | | | .D | | .D | | .W | 1 |
| 14 | 1110 | FSLE | .W | FMAX | .W | MSUBR_Q | .H | 0 |
| | | | .D | | .D | | .W | 1 |
| 15 | 1111 | FSULE | .W | FMAX_A | .W | * | | 0 |
| | | | .D | | .D | | | 1 |

1. See Table 3.11 and Table 3.12.

**Table 3.28 Encoding of Operation Field for BIT Instruction Format**

| **operation** | | *Bits 5...0* | | **data format**[1] | |
|---|---|---|---|---|---|
| | | 9 | 10 | | |
| *Bits 25...23* | | 001001 | 001010 | *Bits 22...16* | |
| 0 | 000 | SLLI | SAT_S | 1110mmm | .B |
| | | | | 110mmmm | .H |
| | | | | 10mmmmm | .W |
| | | | | 0mmmmmm | .D |
| 1 | 001 | SRAI | SAT_U | 1110mmm | .B |
| | | | | 110mmmm | .H |
| | | | | 10mmmmm | .W |
| | | | | 0mmmmmm | .D |
| 2 | 010 | SRLI | SRARI | 1110mmm | .B |
| | | | | 110mmmm | .H |
| | | | | 10mmmmm | .W |
| | | | | 0mmmmmm | .D |
| 3 | 011 | BCLRI | SRLRI | 1110mmm | .B |
| | | | | 110mmmm | .H |
| | | | | 10mmmmm | .W |
| | | | | 0mmmmmm | .D |
| 4 | 100 | BSETI | * | 1110mmm | .B |
| | | | | 110mmmm | .H |
| | | | | 10mmmmm | .W |
| | | | | 0mmmmmm | .D |

**Table 3.28 Encoding of Operation Field for BIT Instruction Format (Continued)**

| | | | | | |
|---|---|---|---|---|---|
| 5 | 101 | BNEGI | * | 1110mmm | .B |
| | | | | 110mmmm | .H |
| | | | | 10mmmmm | .W |
| | | | | 0mmmmmm | .D |
| 6 | 110 | BINSLI | * | 1110mmm | .B |
| | | | | 110mmmm | .H |
| | | | | 10mmmmm | .W |
| | | | | 0mmmmmm | .D |
| 7 | 111 | BINSRI | * | 1110mmm | .B |
| | | | | 110mmmm | .H |
| | | | | 10mmmmm | .W |
| | | | | 0mmmmmm | .D |

1. See Table 3.14.

*Chapter 4*

# The MIPS64® SIMD Architecture Instruction Set

## 4.1 Instruction Set Descriptions

The MIPS64® SIMD Architecture (MSA) consists of integer, fixed-point, and floating-point instructions, all encoded in the MSA major opcode space.

Most MSA instructions operate vector element by vector element in a typical SIMD manner. Few instructions handle the operands as bit vectors because the elements don't make sense, e.g. for the bitwise logical operations.

For certain instructions, the source operand could be an immediate value or a specific vector element selected by an immediate index. The immediate or vector element is being used as a fixed operand across all destination vector elements.

The next two sections list MSA instructions grouped by category and provide individual instruction descriptions arranged in alphabetical order. The constant WRLEN used in all instruction descriptions is set to 128, i.e. the MSA vector register width in bits.

### 4.1.1 Instruction Set Summary by Category

MSA instruction set implements the following categories of instructions: integer arithmetic (Table 4.1), bitwise (Table 4.2), floating-point arithmetic (Table 4.3) and non arithmetic (Table 4.4), floating-point compare (Table 4.5), floating-point conversions (Table 4.6), fixed-point (Table 4.7), branch and compare (Table 4.8), load/store and move (Table 4.9), and element permute (Table 4.10).

The left-shift add instructions LSA and DLSA (Table 4.11) are integral part of the MIPS base architecture. The corresponding documentation pages will be incorporated in the future releases of the base architecture specifications.

**Table 4.1 MSA Integer Arithmetic Instructions**

| Mnemonic | Instruction Description |
|---|---|
| ADDV, ADDVI | Add |
| ADD_A, ADDS_A | Add and Saturated Add Absolute Values |
| ADDS_S, ADDS_U | Signed and Unsigned Saturated Add |
| HADD_S, HADD_U | Signed and Unsigned Horizontal Add |
| ASUB_S, ASUB_U | Absolute Value of Signed and Unsigned Subtract |
| AVE_S, AVE_U | Signed and Unsigned Average |
| AVER_S, AVER_U | Signed and Unsigned Average with Rounding |

**Table 4.1 MSA Integer Arithmetic Instructions (Continued)**

| Mnemonic | Instruction Description |
|---|---|
| DOTP_S, DOTP_U | Signed and Unsigned Dot Product |
| DPADD_S, DPADD_U | Signed and Unsigned Dot Product Add |
| DPSUB_S, DPSUB_U | Signed and Unsigned Dot Product Subtract |
| DIV_S, DIV_U | Divide |
| MADDV | Multiply-Add |
| MAX_A, MIN_A | Maximum and Minimum of Absolute Values |
| MAX_S, MAXI_S, MAX_U, MAXI_U | Signed and Unsigned Maximum |
| MIN_S, MINI_S, MIN_U, MINI_U | Signed and Unsigned Maximum |
| MSUBV | Multiply-Subtract |
| MULV | Multiply |
| MOD_S, MOD_U | Signed and Unsigned Remainder (Modulo) |
| SAT_S, SAT_U | Signed and Unsigned Saturate |
| SUBS_S, SUBS_U | Signed and Unsigned Saturated Subtract |
| HSUB_S, HSUB_U | Signed and Unsigned Horizontal Subtract |
| SUBSUU_S | Signed Saturated Unsigned Subtract |
| SUBSUS_U | Unsigned Saturated Signed Subtract from Unsigned |
| SUBV, SUBVI | Subtract |

**Table 4.2 MSA Bitwise Instructions**

| Mnemonic | Instruction Description |
|---|---|
| AND, ANDI | Logical And |
| BCLR, BCLRI | Bit Clear |
| BINSL, BINSLI, BINSR, BINSRI | Bit Insert Left and Right |
| BMNZ, BMNZI | Bit Move If Not Zero |
| BMZ, BMZI | Bit Move If Zero |
| BNEG, BNEGI | Bit Negate |
| BSEL, BSELI | Bit Select |
| BSET, BSETI | Bit Set |
| NLOC | Leading One Bits Count |

**Table 4.2 MSA Bitwise Instructions (Continued)**

| Mnemonic | Instruction Description |
|---|---|
| NLZC | Leading Zero Bits Count |
| NOR, NORI | Logical Negated Or |
| PCNT | Population (Bits Set to 1) Count |
| OR, ORI | Logical Or |
| SLL, SLLI | Shift Left |
| SRA, SRAI | Shift Right Arithmetic |
| SRAR, SRARI | Rounding Shift Right Arithmetic |
| SRL, SRLI | Shift Right Logical |
| SRLR, SRLRI | Rounding Shift Right Logical |
| XOR, XORI | Logical Exclusive Or |

**Table 4.3 MSA Floating-Point Arithmetic Instructions**

| Mnemonic | Instruction Description |
|---|---|
| FADD | Floating-Point Addition |
| FDIV | Floating-Point Division |
| FEXP2 | Floating-Point Base 2 Exponentiation |
| FLOG2 | Floating-Point Base 2 Logarithm |
| FMADD, FMSUB | Floating-Point Fused Multiply-Add and Multiply-Subtract |
| FMAX, FMIN | Floating-Point Maximum and Minimum |
| FMAX_A, FMIN_A | Floating-Point Maximum and Minimum of Absolute Values |
| FMUL | Floating-Point Multiplication |
| FRCP | Approximate Floating-Point Reciprocal |
| FRINT | Floating-Point Round to Integer |
| FRSQRT | Approximate Floating-Point Reciprocal of Square Root |
| FSQRT | Floating-Point Square Root |
| FSUB | Floating-Point Subtraction |

### Table 4.4 MSA Floating-Point Non Arithmetic Instructions

| Mnemonic | Instruction Description |
|---|---|
| FCLASS | Floating-Point Class Mask |

### Table 4.5 MSA Floating-Point Compare Instructions

| Mnemonic | Instruction Description |
|---|---|
| FCAF | Floating-Point Quiet Compare Always False |
| FCUN | Floating-Point Quiet Compare Unordered |
| FCOR | Floating-Point Quiet Compare Ordered |
| FCEQ | Floating-Point Quiet Compare Equal |
| FCUNE | Floating-Point Quiet Compare Unordered or Not Equal |
| FCUEQ | Floating-Point Quiet Compare Unordered or Equal |
| FCNE | Floating-Point Quiet Compare Not Equal |
| FCLT | Floating-Point Quiet Compare Less Than |
| FCULT | Floating-Point Quiet Compare Unordered or Less Than |
| FCLE | Floating-Point Quiet Compare Less Than or Equal |
| FCULE | Floating-Point Quiet Compare Unordered or Less Than or Equal |
| FSAF | Floating-Point Signaling Compare Always False |
| FSUN | Floating-Point Signaling Compare Unordered |
| FSOR | Floating-Point Signaling Compare Ordered |
| FSEQ | Floating-Point Signaling Compare Equal |
| FSUNE | Floating-Point Signaling Compare Unordered or Not Equal |
| FSUEQ | Floating-Point Signaling Compare Unordered or Equal |
| FSNE | Floating-Point Signaling Compare Not Equal |
| FSLT | Floating-Point Signaling Compare Less Than |
| FSULT | Floating-Point Signaling Compare Unordered or Less Than |
| FSLE | Floating-Point Signaling Compare Less Than or Equal |
| FSULE | Floating-Point Signaling Compare Unordered or Less Than or Equal |

### Table 4.6 MSA Floating-Point Conversion Instructions

| Mnemonic | Instruction Description |
|---|---|
| FEXDO | Floating-Point Down-Convert Interchange Format |
| FEXUPL, FEXUPR | Left-Half and Right-Half Floating-Point Up-Convert Interchange Format |
| FFINT_S, FFINT_U | Floating-Point Convert from Signed and Unsigned Integer |
| FFQL, FFQR | Left-Half and Right-Half Floating-Point Convert from Fixed-Point |
| FTINT_S, FTINT_U | Floating-Point Round and Convert to Signed and Unsigned Integer |
| FTRUNC_S, FTRUNC_U | Floating-Point Truncate and Convert to Signed and Unsigned Integer |
| FTQ | Floating-Point Round and Convert to Fixed-Point |

### Table 4.7 MSA Fixed-Point Instructions

| Mnemonic | Instruction Description |
|---|---|
| MADD_Q, MADDR_Q | Fixed-Point Multiply and Add without and with Rounding |
| MSUB_Q, MSUBR_Q | Fixed-Point Multiply and Subtract without and with Rounding |
| MUL_Q, MULR_Q | Fixed-Point Multiply without and with Rounding |

### Table 4.8 MSA Branch and Compare Instructions

| Mnemonic | Instruction Description |
|---|---|
| BNZ | Branch If Not Zero |
| BZ | Branch If Zero |
| CEQ, CEQI | Compare Equal |
| CLE_S, CLEI_S, CLE_U, CLEI_U | Compare Less-Than-or-Equal Signed and Unsigned |
| CLT_S, CLTI_S, CLT_U, CLTI_U | Compare Less-Than Signed and Unsigned |

**Table 4.9 MSA Load/Store and Move Instructions**

| Mnemonic | Instruction Description |
|---|---|
| CFCMSA, CTCMSA | Copy from and copy to MSA Control Register |
| LD | Load Vector |
| LDI | Load Immediate |
| MOVE | Vector to Vector Move |
| SPLAT, SPLATI | Replicate Vector Element |
| FILL | Fill Vector from GPR |
| INSERT, INSVE | Insert GPR and Vector element 0 to Vector Element |
| COPY_S, COPY_U | Copy element to GPR Signed and Unsigned |
| ST | Store Vector |

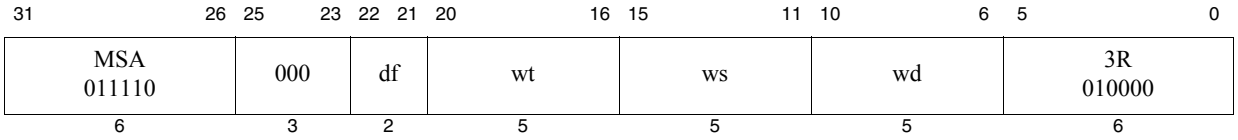
**Table 4.10 MSA Element Permute Instructions**

| Mnemonic | Instruction Description |
|---|---|
| ILVEV, ILVOD | Interleave Even, Odd |
| ILVL, ILVR | Interleave the Left, Right |
| PCKEV, PCKOD | Pack Even and Odd Elements |
| SHF | Set Shuffle |
| SLD, SLDI | Element Slide |
| VSHF | Vector shuffle |

**Table 4.11 Base Architecture Instructions**

| Mnemonic | Instruction Description |
|---|---|
| LSA | Left-shift add or load/store address calculation. |
| DLSA | Double left-shift add or load/store address calculation. |

### 4.1.2 Alphabetical List of Instructions




**Format:** `ADD_A.df`

`ADD_A.B wd,ws,wt` **MSA**

`ADD_A.H wd,ws,wt` **MSA**

`ADD_A.W wd,ws,wt` **MSA**

`ADD_A.D wd,ws,wt` **MSA**

**Purpose:** Vector Add Absolute Values

Vector addition to vector using the absolute values.

**Description:** `wd[i] ← absolute_value(ws[i]) + absolute_value(wt[i])`

The absolute values of the elements in vector *wt* are added to the absolute values of the elements in vector *ws*. The result is written to vector *wd*.

The operands and results are values in integer data format *df*.

**Restrictions:**

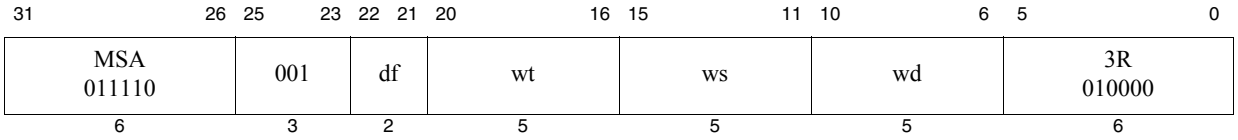
No data-dependent exceptions are possible.

**Operation:**

```
ADD_A.B
    for i in 0 .. WRLEN/8-1
        WR[wd]8i+7..8i ← abs(WR[ws]8i+7..8i, 8) + abs(WR[wt]8i+7..8i, 8)
    endfor

ADD_A.H
    for i in 0 .. WRLEN/16-1
        WR[wd]16i+15..16i ← abs(WR[ws]16i+15..16i, 16) + abs(WR[wt]16i+15..16i, 16)
    endfor

ADD_A.W
    for i in 0 .. WRLEN/32-1
        WR[wd]32i+31..32i ← abs(WR[ws]32i+31..32i, 32) + abs(WR[wt]32i+31..32i, 32)
    endfor

ADD_A.D
    for i in 0 .. WRLEN/64-1
        WR[wd]64i+63..64i ← abs(WR[ws]64i+63..64i, 64) + abs(WR[wt]64i+63..64i, 64)
    endfor

function abs(tt, n)
    if ttn-1 = 1 then
        return -ttn-1...0
    else
        return ttn-1..0
    endif
endfunction abs
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception.




**Format:** `ADDS_A.df`

`ADDS_A.B wd,ws,wt` **MSA**

`ADDS_A.H wd,ws,wt` **MSA**

`ADDS_A.W wd,ws,wt` **MSA**

`ADDS_A.D wd,ws,wt` **MSA**

**Purpose:** Vector Saturated Add of Absolute Values

Vector saturated addition to vector of absolute values.

**Description:** `wd[i] ← saturate_signed(absolute_value(ws[i]) + absolute_value(wt[i]))`

The absolute values of the elements in vector *wt* are added to the absolute values of the elements in vector *ws*. The saturated signed result is written to vector *wd*.

The operands and results are values in integer data format *df*.

**Restrictions:**

No data-dependent exceptions are possible.

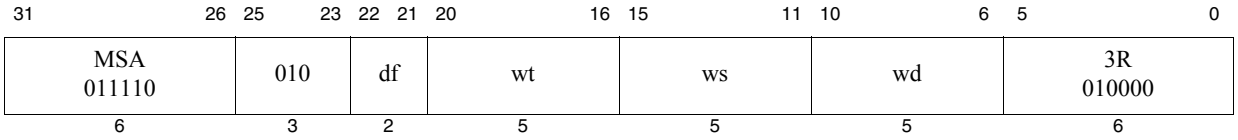
**Operation:**

```
ADDS_A.B
    for i in 0 .. WRLEN/8-1
        WR[wd]8i+7..8i ← adds_a(WR[ws]8i+7..8i, WR[wt]8i+7..8i, 8)
    endfor

ADDS_A.H
    for i in 0 .. WRLEN/16-1
        WR[wd]16i+15..16i ← adds_a(WR[ws]16i+15..16i, WR[wt]16i+15..16i, 16)
    endfor

ADDS_A.W
    for i in 0 .. WRLEN/32-1
        WR[wd]32i+31..32i ← adds_a(WR[ws]32i+31..32i, WR[wt]32i+31..32i, 32)
    endfor

ADDS_A.D
    for i in 0 .. WRLEN/64-1
        WR[wd]64i+63..64i ← adds_a(WR[ws]64i+63..64i, WR[wt]64i+63..64i, 64)
    endfor

function abs(tt, n)
    if ttn-1 = 1 then
        return -ttn-1...0
    else
        return ttn-1..0
    endif
endfunction abs

function sat_s(tt, n, b)
    if ttn-1 = 0 and ttn-1..b-1 ≠ 0n-b+1 then
```

```
        return 0^(n-b+1) || 1^(b-1)
    endif
    if tt_(n-1) = 1 and tt_(n-1..b-1) ≠ 1^(n-b+1) then
        return 1^(n-b+1) || 0^(b-1)
    else
        return tt
    endif
endfunction sat_s

function adds_a(ts, tt, n)
    t ← (0 || abs(ts, n)) + (0 || abs(tt, n))
    return sat_s(t, n+1, n)
endfunction adds_a
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception.




**Format:** `ADDS_S.df`

`ADDS_S.B wd,ws,wt` **MSA**

`ADDS_S.H wd,ws,wt` **MSA**

`ADDS_S.W wd,ws,wt` **MSA**

`ADDS_S.D wd,ws,wt` **MSA**

**Purpose:** Vector Signed Saturated Add of Signed Values

Vector addition to vector saturating the result as signed value.

**Description:** `wd[i] ← saturate_signed(signed(ws[i]) + signed(wt[i]))`

The elements in vector *wt* are added to the elements in vector *ws*. Signed arithmetic is performed and overflows clamp to the largest and/or smallest representable signed values before writing the result to vector *wd*.

The operands and results are values in integer data format *df*.

**Restrictions:**

No data-dependent exceptions are possible.

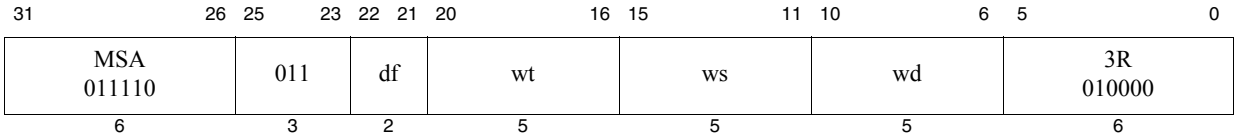
**Operation:**

```
ADDS_S.B
    for i in 0 .. WRLEN/8-1
        WR[wd]8i+7..8i ← adds_s(WR[ws]8i+7..8i, WR[wt]8i+7..8i, 8)
    endfor

ADDS_S.H
    for i in 0 .. WRLEN/16-1
        WR[wd]16i+15..16i ← adds_s(WR[ws]16i+15..16i, WR[wt]16i+15..16i, 16)
    endfor

ADDS_S.W
    for i in 0 .. WRLEN/32-1
        WR[wd]32i+31..32i ← adds_s(WR[ws]32i+31..32i, WR[wt]32i+31..32i, 32)
    endfor

ADDS_S.D
    for i in 0 .. WRLEN/64-1
        WR[wd]64i+63..64i ← adds_s(WR[ws]64i+63..64i, WR[wt]64i+63..64i, 64)
    endfor

function sat_s(tt, n, b)
    if ttn-1 = 0 and ttn-1..b-1 ≠ 0n-b+1 then
        return 0n-b+1 || 1b-1
    endif
    if ttn-1 = 1 and ttn-1..b-1 ≠ 1n-b+1 then
        return 1n-b+1 || 0b-1
    else
        return tt
    endif
endfunction sat_s
```

```
function adds_s(ts, tt, n)
   t ← (ts_{n-1} || ts) + (tt_{n-1} || tt)
   return sat_s(t, n+1, n)
endfunction adds_s
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception.




**Format:** `ADDS_U.df`

`ADDS_U.B wd,ws,wt` **MSA**

`ADDS_U.H wd,ws,wt` **MSA**

`ADDS_U.W wd,ws,wt` **MSA**

`ADDS_U.D wd,ws,wt` **MSA**

**Purpose:** Vector Unsigned Saturated Add of Unsigned Values

Vector addition to vector saturating the result as unsigned value.

**Description:** `wd[i] ← saturate_unsigned(unsigned(ws[i]) + unsigned(wt[i]))`

The elements in vector *wt* are added to the elements in vector *ws*. Unsigned arithmetic is performed and overflows clamp to the largest representable unsigned value before writing the result to vector *wd*.

The operands and results are values in integer data format *df*.

**Restrictions:**

No data-dependent exceptions are possible.

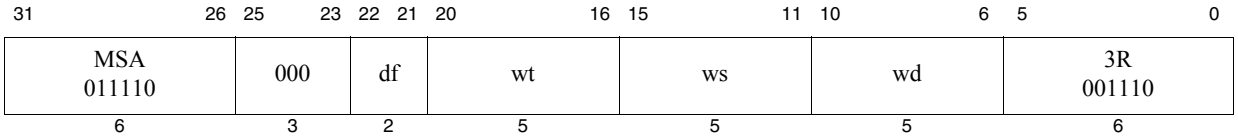
**Operation:**

```
ADDS_U.B
    for i in 0 .. WRLEN/8-1
        WR[wd]8i+7..8i ← adds_u(WR[ws]8i+7..8i, WR[wt]8i+7..8i, 8)
    endfor

ADDS_U.H
    for i in 0 .. WRLEN/16-1
        WR[wd]16i+15..16i ← adds_u(WR[ws]16i+15..16i, WR[wt]16i+15..16i, 16)
    endfor

ADDS_U.W
    for i in 0 .. WRLEN/32-1
        WR[wd]32i+31..32i ← adds_u(WR[ws]32i+31..32i, WR[wt]32i+31..32i, 32)
    endfor

ADDS_U.D
    for i in 0 .. WRLEN/64-1
        WR[wd]64i+63..64i ← adds_u(WR[ws]64i+63..64i, WR[wt]64i+63..64i, 64)
    endfor

function sat_u(tt, n, b)
    if ttn-1..b ≠ 0n-b then
        return 0n-b || 1b
    else
        return tt
    endif
endfunction sat_u

function adds_u(ts, tt, n)
    t ← (0 || ts) + (0 || tt)
```

```
        return sat_u(t, n+1, n)
    endfunction adds_u
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception.




**Format:** `ADDV.df`

`ADDV.B wd,ws,wt` **MSA**

`ADDV.H wd,ws,wt` **MSA**

`ADDV.W wd,ws,wt` **MSA**

`ADDV.D wd,ws,wt` **MSA**

**Purpose:** Vector Add

Vector addition to vector.

**Description:** `wd[i] ← ws[i] + wt[i]`

The elements in vector *wt* are added to the elements in vector *ws*. The result is written to vector *wd*.

The operands and results are values in integer data format *df*.

**Restrictions:**

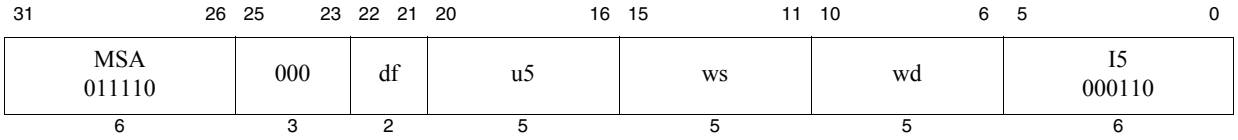
No data-dependent exceptions are possible.

**Operation:**

```
ADDV.B
    for i in 0 .. WRLEN/8-1
        WR[wd]8i+7..8i ← WR[ws]8i+7..8i + WR[wt]8i+7..8i
    endfor

ADDV.H
    for i in 0 .. WRLEN/16-1
        WR[wd]16i+15..16i ← WR[ws]16i+15..16i + WR[wt]16i+15..16i
    endfor

ADDV.W
    for i in 0 .. WRLEN/32-1
        WR[wd]32i+31..32i ← WR[ws]32i+31..32i + WR[wt]32i+31..32i
    endfor

ADDV.D
    for i in 0 .. WRLEN/64-1
        WR[wd]64i+63..64i ← WR[ws]64i+63..64i + WR[wt]64i+63..64i
    endfor
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception.




**Format:** `ADDVI.df`

`ADDVI.B wd,ws,u5` **MSA**

`ADDVI.H wd,ws,u5` **MSA**

`ADDVI.W wd,ws,u5` **MSA**

`ADDVI.D wd,ws,u5` **MSA**

**Purpose:** Immediate Add

Immediate addition to vector.

**Description:** `wd[i] ← ws[i] + u5`

The 5-bit immediate unsigned value *u5* is added to the elements in vector *ws*. The result is written to vector *wd*.

The operands and results are values in integer data format *df*.

**Restrictions:**

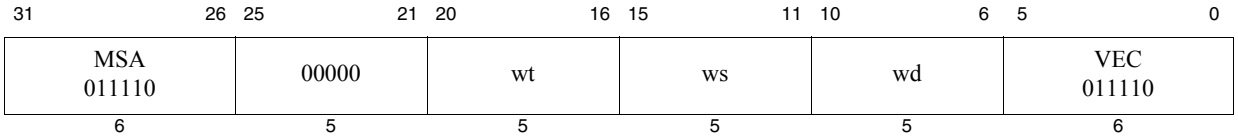
No data-dependent exceptions are possible.

**Operation:**

```
ADDVI.B
    t ← 0^3 || u5_4..0
    for i in 0 .. WRLEN/8-1
        WR[wd]_8i+7..8i ← WR[ws]_8i+7..8i + t
    endfor

ADDVI.H
    t ← 0^11 || u5_4..0
    for i in 0 .. WRLEN/16-1
        WR[wd]_16i+15..16i ← WR[ws]_16i+15..16i + t
    endfor

ADDVI.W
    t ← 0^27 || u5_4..0
    for i in 0 .. WRLEN/32-1
        WR[wd]_32i+31..32i ← WR[ws]_32i+31..32i + t
    endfor

ADDVI.D
    t ← 0^59 || u5_4..0
    for i in 0 .. WRLEN/64-1
        WR[wd]_64i+63..64i ← WR[ws]_64i+63..64i + t
    endfor
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception.




**Format:** `AND.V`
`AND.V wd,ws,wt` **MSA**

**Purpose:** Vector Logical And

Vector by vector logical and.

**Description:** `wd ← ws AND wt`

Each bit of vector *ws* is combined with the corresponding bit of vector *wt* in a bitwise logical AND operation. The result is written to vector *wd*.

The operands and results are bit vector values.

**Restrictions:**

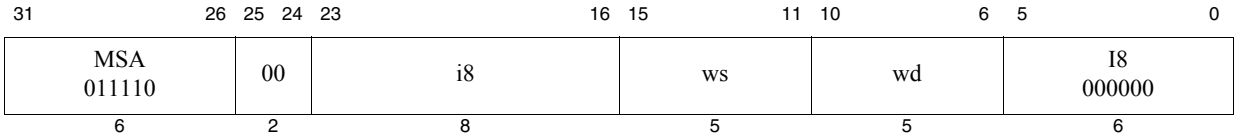
No data-dependent exceptions are possible.

**Operation:**

```
WR[wd] ← WR[ws] and WR[wt]
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception.




**Format:** `ANDI.B`
`ANDI.B wd,ws,i8` **MSA**

**Purpose:** Immediate Logical And

Immediate by vector logical and.

**Description:** `wd[i] ← ws[i] AND i8`

Each byte element of vector *ws* is combined with the 8-bit immediate *i8* in a bitwise logical AND operation. The result is written to vector *wd*.

The operands and results are values in integer byte data format.

**Restrictions:**

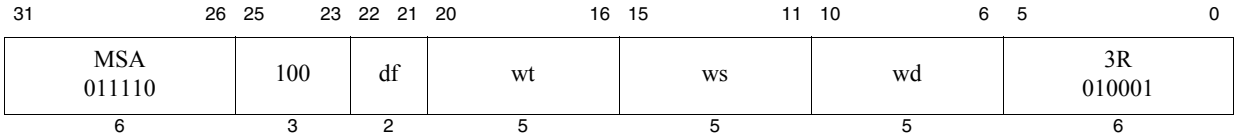
No data-dependent exceptions are possible.

**Operation:**

```
for i in 0 .. WRLEN/8-1
  WR[wd]8i+7..8i ← WR[ws]8i+7..8i and i87..0
endfor
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception.




**Format:** `ASUB_S.df`

`ASUB_S.B wd,ws,wt` **MSA**

`ASUB_S.H wd,ws,wt` **MSA**

`ASUB_S.W wd,ws,wt` **MSA**

`ASUB_S.D wd,ws,wt` **MSA**

**Purpose:** Vector Absolute Values of Signed Subtract

Vector subtraction from vector of signed values taking the absolute value of the results.

**Description:** `wd[i] ← absolute_value(signed(ws[i]) - signed(wt[i]))`

The signed elements in v ector *wt* are subtracted from the signed elements in v ector *ws*. The absolute value of the signed result is written to vector *wd*.

The operands and results are values in integer data format *df*.

**Restrictions:**

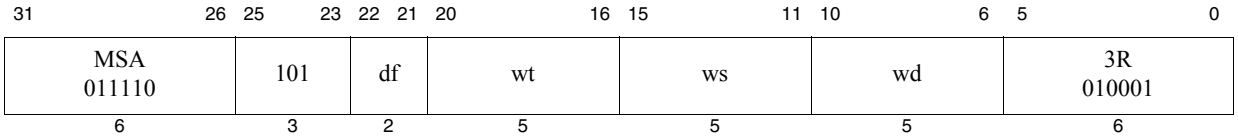
No data-dependent exceptions are possible.

**Operation:**

```
ASUB_S.B
    for i in 0 .. WRLEN/8-1
        WR[wd]8i+7..8i ← asub_s(WR[ws]8i+7..8i, WR[wt]8i+7..8i, 8)
    endfor

ASUB_S.H
    for i in 0 .. WRLEN/16-1
        WR[wd]16i+15..16i ← asub_s(WR[ws]16i+15..16i, WR[wt]16i+15..16i, 16)
    endfor

ASUB_S.W
    for i in 0 .. WRLEN/32-1
        WR[wd]32i+31..32i ← asub_s(WR[ws]32i+31..32i, WR[wt]32i+31..32i, 32)
    endfor

ASUB_S.D
    for i in 0 .. WRLEN/64-1
        WR[wd]64i+63..64i ← asub_s(WR[ws]64i+63..64i, WR[wt]64i+63..64i, 64)
    endfor

function asub_s(ts, tt, n)
    t ← (tsn-1 || ts) - (ttn-1 || tt)
    if tn = 0 then
        return tn-1..0
    else
        return (-t)n-1..0
endfunction asub_s
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception.




**Format:** `ASUB_U.df`

`ASUB_U.B wd,ws,wt` **MSA**

`ASUB_U.H wd,ws,wt` **MSA**

`ASUB_U.W wd,ws,wt` **MSA**

`ASUB_U.D wd,ws,wt` **MSA**

**Purpose:** Vector Absolute Values of Unsigned Subtract

Vector subtraction from vector of unsigned values taking the absolute value of the results.

**Description:** `wd[i] ← absolute_value(unsigned(ws[i]) - unsigned(wt[i]))`

The unsigned elements in vector *wt* are subtracted from the unsigned elements in vector *ws*. The absolute value of the signed result is written to vector *wd*.

The operands and results are values in integer data format *df*.

**Restrictions:**

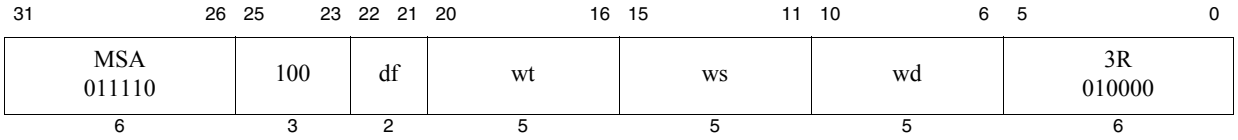
No data-dependent exceptions are possible.

**Operation:**

```
ASUB_U.B
    for i in 0 .. WRLEN/8-1
        WR[wd]8i+7..8i ← asub_u(WR[ws]8i+7..8i, WR[wt]8i+7..8i, 8)
    endfor

ASUB_U.H
    for i in 0 .. WRLEN/16-1
        WR[wd]16i+15..16i ← asub_u(WR[ws]16i+15..16i, WR[wt]16i+15..16i, 16)
    endfor

ASUB_U.W
    for i in 0 .. WRLEN/32-1
        WR[wd]32i+31..32i ← asub_u(WR[ws]32i+31..32i, WR[wt]32i+31..32i, 32)
    endfor

ASUB_U.D
    for i in 0 .. WRLEN/64-1
        WR[wd]64i+63..64i ← asub_u(WR[ws]64i+63..64i, WR[wt]64i+63..64i, 64)
    endfor

function asub_u(ts, tt, n)
    t ← (0 || ts) - (0 || tt)
    if tn = 0 then
        return tn-1..0
    else
        return (-t)n-1..0
endfunction asub_s
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception.




**Format:** `AVE_S.df`

`AVE_S.B wd,ws,wt` **MSA**

`AVE_S.H wd,ws,wt` **MSA**

`AVE_S.W wd,ws,wt` **MSA**

`AVE_S.D wd,ws,wt` **MSA**

**Purpose:** Vector Signed Average

Vector average using the signed values.

**Description:** `wd[i] ← (ws[i] + wt[i]) / 2`

The elements in vector *wt* are added to the elements in vector *ws*. The addition is done signed with full precision, i.e. the result has one extra bit. Signed division by 2 (or arithmetic shift right by one bit) is performed before writing the result to vector *wd*.

The operands and results are values in integer data format *df*.

**Restrictions:**

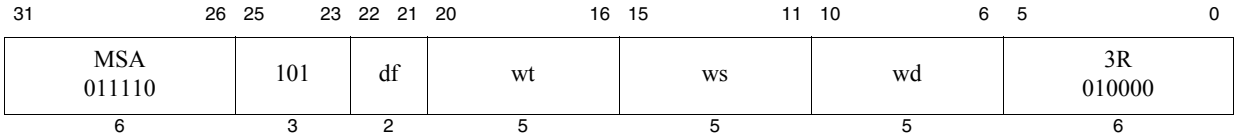
No data-dependent exceptions are possible.

**Operation:**

```
AVE_S.B
    for i in 0 .. WRLEN/8-1
        WR[wd]8i+7..8i ← ave_s(WR[ws]8i+7..8i, WR[wt]8i+7..8i, 8)
    endfor

AVE_S.H
    for i in 0 .. WRLEN/16-1
        WR[wd]16i+15..16i ← ave_s(WR[ws]16i+15..16i, WR[wt]16i+15..16i, 16)
    endfor

AVE_S.W
    for i in 0 .. WRLEN/32-1
        WR[wd]32i+31..32i ← ave_s(WR[ws]32i+31..32i, WR[wt]32i+31..32i, 32)
    endfor

AVE_S.D
    for i in 0 .. WRLEN/64-1
        WR[wd]64i+63..64i ← ave_s(WR[ws]64i+63..64i, WR[wt]64i+63..64i, 64)
    endfor

function ave_s(ts, tt, n)
    t ← (tsn-1 || ts) + (ttn-1 || tt)
    return tn..1
endfunction ave_s
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception.




**Format:** `AVE_U.df`

`AVE_U.B wd,ws,wt` **MSA**

`AVE_U.H wd,ws,wt` **MSA**

`AVE_U.W wd,ws,wt` **MSA**

`AVE_U.D wd,ws,wt` **MSA**

**Purpose:** Vector Unsigned Average

Vector average using the unsigned values.

**Description:** `wd[i] ← (ws[i] + wt[i]) / 2`

The elements in vector *wt* are added to the elements in vector *ws*. The addition is done unsigned with full precision, i.e. the result has one extra bit. Unsigned division by 2 (or logical shift right by one bit) is performed before writing the result to vector *wd*.

The operands and results are values in integer data format *df*.

**Restrictions:**

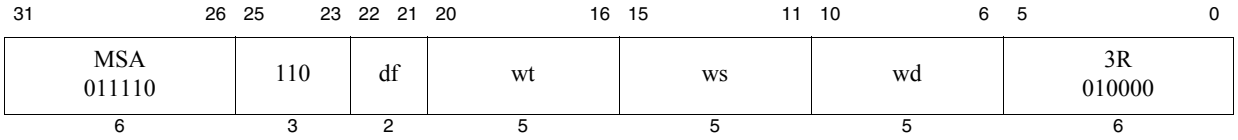
No data-dependent exceptions are possible.

**Operation:**

```
AVE_U.B
    for i in 0 .. WRLEN/8-1
        WR[wd]8i+7..8i ← ave_u(WR[ws]8i+7..8i, WR[wt]8i+7..8i, 8)
    endfor

AVE_U.H
    for i in 0 .. WRLEN/16-1
        WR[wd]16i+15..16i ← ave_u(WR[ws]16i+15..16i, WR[wt]16i+15..16i, 16)
    endfor

AVE_U.W
    for i in 0 .. WRLEN/32-1
        WR[wd]32i+31..32i ← ave_u(WR[ws]32i+31..32i, WR[wt]32i+31..32i, 32)
    endfor

AVE_U.D
    for i in 0 .. WRLEN/64-1
        WR[wd]64i+63..64i ← ave_u(WR[ws]64i+63..64i, WR[wt]64i+63..64i, 64)
    endfor

function ave_u(ts, tt, n)
    t ← (0 || ts) + (0 || tt)
    return tn..1
endfunction ave_u
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception.




**Format:** `AVER_S.df`

`AVER_S.B wd,ws,wt` **MSA**

`AVER_S.H wd,ws,wt` **MSA**

`AVER_S.W wd,ws,wt` **MSA**

`AVER_S.D wd,ws,wt` **MSA**

**Purpose:** Vector Signed Average Rounded

Vector average rounded using the signed values.

**Description:** `wd[i] ← (ws[i] + wt[i] + 1) / 2`

The elements in vector *wt* are added to the elements in vector *ws*. The addition of the elements plus 1 (for rounding) is done signed with full precision, i.e. the result has one extra bit. Signed division by 2 (or arithmetic shift right by one bit) is performed before writing the result to vector *wd*.

The operands and results are values in integer data format *df*.

**Restrictions:**

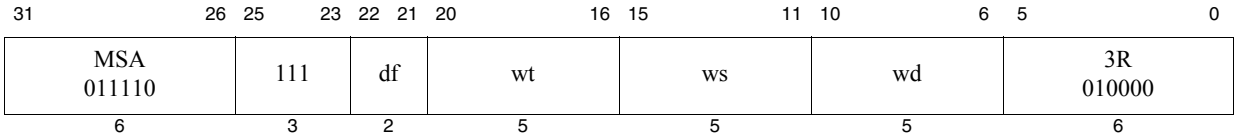
No data-dependent exceptions are possible.

**Operation:**

```
AVER_S.B
    for i in 0 .. WRLEN/8-1
        WR[wd]8i+7..8i ← aver_s(WR[ws]8i+7..8i, WR[wt]8i+7..8i, 8)
    endfor

AVER_S.H
    for i in 0 .. WRLEN/16-1
        WR[wd]16i+15..16i ← aver_s(WR[ws]16i+15..16i, WR[wt]16i+15..16i, 16)
    endfor

AVER_S.W
    for i in 0 .. WRLEN/32-1
        WR[wd]32i+31..32i ← aver_s(WR[ws]32i+31..32i, WR[wt]32i+31..32i, 32)
    endfor

AVER_S.D
    for i in 0 .. WRLEN/64-1
        WR[wd]64i+63..64i ← aver_s(WR[ws]64i+63..64i, WR[wt]64i+63..64i, 64)
    endfor

function ave_s(ts, tt, n)
    t ← (tsn-1 || ts) + (ttn-1 || tt) + 1
    return tn..1
endfunction aver_s
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception.




**Format:** `AVER_U.df`

`AVER_U.B wd,ws,wt` **MSA**

`AVER_U.H wd,ws,wt` **MSA**

`AVER_U.W wd,ws,wt` **MSA**

`AVER_U.D wd,ws,wt` **MSA**

**Purpose:** Vector Unsigned Average Rounded

Vector average rounded using the unsigned values.

**Description:** `wd[i] ← (ws[i] + wt[i] + 1) / 2`

The elements in vector *wt* are added to the elements in vector *ws*. The addition of the elements plus 1 (for rounding) is done unsigned with full precision, i.e. the result has one extra bit. Unsigned division by 2 (or logical shift right by one bit) is performed before writing the result to vector *wd*.

The operands and results are values in integer data format *df*.

**Restrictions:**

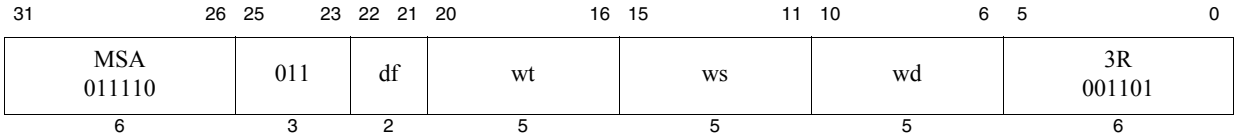
No data-dependent exceptions are possible.

**Operation:**

```
AVER_U.B
    for i in 0 .. WRLEN/8-1
        WR[wd]8i+7..8i ← aver_u(WR[ws]8i+7..8i, WR[wt]8i+7..8i, 8)
    endfor

AVER_U.H
    for i in 0 .. WRLEN/16-1
        WR[wd]16i+15..16i ← aver_u(WR[ws]16i+15..16i, WR[wt]16i+15..16i, 16)
    endfor

AVER_U.W
    for i in 0 .. WRLEN/32-1
        WR[wd]32i+31..32i ← aver_u(WR[ws]32i+31..32i, WR[wt]32i+31..32i, 32)
    endfor

AVER_U.D
    for i in 0 .. WRLEN/64-1
        WR[wd]64i+63..64i ← aver_u(WR[ws]64i+63..64i, WR[wt]64i+63..64i, 64)
    endfor

function ave_u(ts, tt, n)
    t ← (0 || ts) + (0 || tt) + 1
    return tn..1
endfunction aver_u
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception.




**Format:** `BCLR.df`

`BCLR.B wd,ws,wt` **MSA**

`BCLR.H wd,ws,wt` **MSA**

`BCLR.W wd,ws,wt` **MSA**

`BCLR.D wd,ws,wt` **MSA**

**Purpose:** Vector Bit Clear

Vector selected bit position clear in each element.

**Description:** `wd[i] ← bit_clear(ws[i], wt[i])`

Clear (set to 0) one bit in each element of vector *ws*. The bit position is given by the elements in *wt* modulo the size of the element in bits. The result is written to vector *wd*.

The operands and results are values in integer data format *df*.

**Restrictions:**

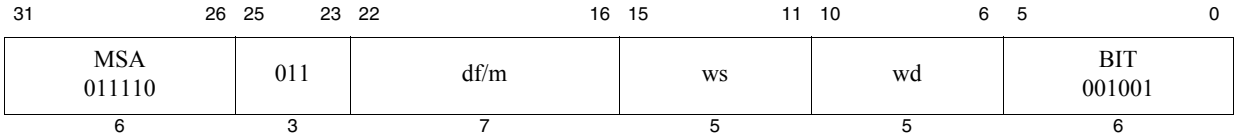
No data-dependent exceptions are possible.

**Operation:**

```
BCLR.B
    for i in 0 .. WRLEN/8-1
        t ← WR[wt]8i+2..8i
        WR[wd]8i+7..8i ← WR[ws]8i+7..8i and (1^(7-t) || 0 || 1^t)
    endfor

BCLR.H
    for i in 0 .. WRLEN/16-1
        t ← WR[wt]16i+3..16i
        WR[wd]16i+15..16i ← WR[ws]16i+15..16i and (1^(15-t) || 0 || 1^t)
    endfor

BCLR.W
    for i in 0 .. WRLEN/32-1
        t ← WR[wt]32i+4..32i
        WR[wd]32i+31..32i ← WR[ws]32i+31..32i and (1^(31-t) || 0 || 1^t)
    endfor

BCLR.D
    for i in 0 .. WRLEN/64-1
        t ← WR[wt]64i+5..64i
        WR[wd]64i+63..64i ← WR[ws]64i+63..64i and (1^(63-t) || 0 || 1^t)
    endfor
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception.




**Format:** `BCLRI.df`

`BCLRI.B wd,ws,m` **MSA**

`BCLRI.H wd,ws,m` **MSA**

`BCLRI.W wd,ws,m` **MSA**

`BCLRI.D wd,ws,m` **MSA**

**Purpose:** Immediate Bit Clear

Immediate selected bit position clear in each element.

**Description:** `wd[i] ← bit_clear(ws[i], m)`

Clear (set to 0) one bit in each element of vector *ws*. The bit position is given by the immediate *m* modulo the size of the element in bits. The result is written to vector *wd*.

The operands and results are values in integer data format *df*.

**Restrictions:**

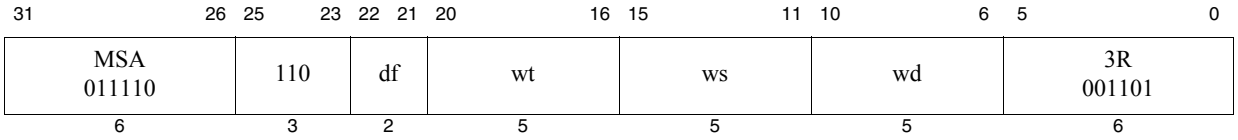
No data-dependent exceptions are possible.

**Operation:**

```
BCLRI.B
   t ← m
   for i in 0 .. WRLEN/8-1
      WR[wd]8i+7..8i ← WR[ws]8i+7..8i and (1^(7-t) || 0 || 1^t)
   endfor

BCLRI.H
   t ← m
   for i in 0 .. WRLEN/16-1
      WR[wd]16i+15..16i ← WR[ws]16i+15..16i and (1^(15-t) || 0 || 1^t)
   endfor

BCLRI.W
   t ← m
   for i in 0 .. WRLEN/32-1
      WR[wd]32i+31..32i ← WR[ws]32i+31..32i and (1^(31-t) || 0 || 1^t)
   endfor

BCLRI.D
   t ← m
   for i in 0 .. WRLEN/64-1
      WR[wd]64i+63..64i ← WR[ws]64i+63..64i and (1^(63-t) || 0 || 1^t)
   endfor
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception.




**Format:** `BINSL.df`

`BINSL.B wd,ws,wt` **MSA**

`BINSL.H wd,ws,wt` **MSA**

`BINSL.W wd,ws,wt` **MSA**

`BINSL.D wd,ws,wt` **MSA**

**Purpose:** Vector Bit Insert Left

Vector selected left most bits copy while preserving destination right bits.

**Description:** `wd[i] ← bit_insert_left(wd[i], ws[i], wt[i])`

Copy most significant (left) bits in each element of vector *ws* to elements in vector *wd* while preserving the least significant (right) bits. The number of bits to copy is given by the elements in vector *wt* modulo the size of the element in bits plus 1.

The operands and results are values in integer data format *df*.

**Restrictions:**

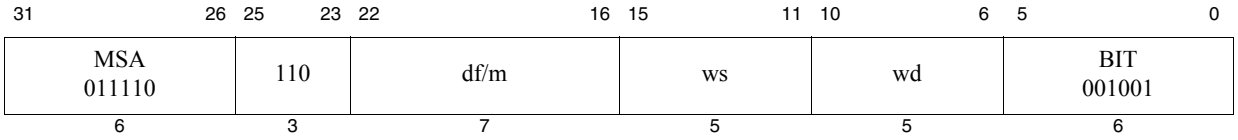
No data-dependent exceptions are possible.

**Operation:**

```
BINSL.B
    for i in 0 .. WRLEN/8-1
        t ← WR[wt]8i+2..8i
        WR[wd]8i+7..8i ← WR[ws]8i+7..8i+7-t || WR[wd]8i+7-t-1..8i
    endfor

BINSL.H
    for i in 0 .. WRLEN/16-1
        t ← WR[wt]16i+3..16i
        WR[wd]16i+15..16i ← WR[ws]16i+15..16i+15-t || WR[wd]16i+15-t-1..16i
    endfor

BINSL.W
    for i in 0 .. WRLEN/32-1
        t ← WR[wt]32i+4..32i
        WR[wd]32i+31..32i ← WR[ws]32i+31..32i+31-t || WR[wd]32i+31-t-1..32i
    endfor

BINSL.D
    for i in 0 .. WRLEN/64-1
        t ← WR[wt]64i+5..64i
        WR[wd]64i+63..64i ← WR[ws]64i+63..64i+63-t || WR[wd]64i+63-t-1..64i
    endfor
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception.




**Format:** `BINSLI.df`

`BINSLI.B wd,ws,m` **MSA**

`BINSLI.H wd,ws,m` **MSA**

`BINSLI.W wd,ws,m` **MSA**

`BINSLI.D wd,ws,m` **MSA**

**Purpose:** Immediate Bit Insert Left

Immediate selected left most bits copy while preserving destination right bits.

**Description:** `wd[i] ← bit_insert_left(wd[i], ws[i], m)`

Copy most significant (left) bits in each element of vector *ws* to elements in vector *wd* while preserving the least significant (right) bits. The number of bits to copy is given by the immediate *m* modulo the size of the element in bits plus 1.

The operands and results are values in integer data format *df*.

**Restrictions:**

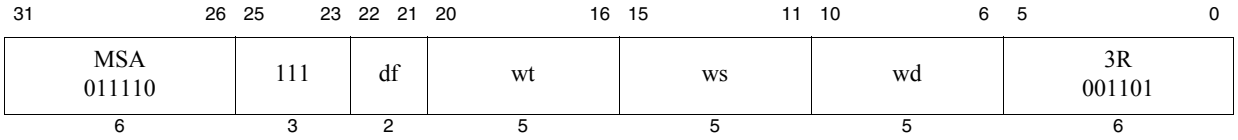
No data-dependent exceptions are possible.

**Operation:**

```
BINSLI.B
    t ← m
    for i in 0 .. WRLEN/8-1
        WR[wd]8i+7..8i ← WR[ws]8i+7..8i+7-t || WR[wd]8i+7-t-1..8i
    endfor

BINSLI.H
    t ← m
    for i in 0 .. WRLEN/16-1
        WR[wd]16i+15..16i ← WR[ws]16i+15..16i+15-t || WR[wd]16i+15-t-1..16i
    endfor

BINSLI.W
    t ← m
    for i in 0 .. WRLEN/32-1
        WR[wd]32i+31..32i ← WR[ws]32i+31..32i+31-t || WR[wd]32i+31-t-1..32i
    endfor

BINSLI.D
    t ← m
    for i in 0 .. WRLEN/64-1
        WR[wd]64i+63..64i ← WR[ws]64i+63..64i+63-t || WR[wd]64i+63-t-1..64i
    endfor
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception.




**Format:** `BINSR.df`

`BINSR.B wd,ws,wt` **MSA**

`BINSR.H wd,ws,wt` **MSA**

`BINSR.W wd,ws,wt` **MSA**

`BINSR.D wd,ws,wt` **MSA**

**Purpose:** Vector Bit Insert Right

Vector selected right most bits copy while preserving destination left bits.

**Description:** `wd[i] ← bit_insert_right(wd[i], ws[i], wt[i])`

Copy least significant (right) bits in each element of vector *ws* to elements in vector *wd* while preserving the most significant (left) bits. The number of bits to copy is given by the elements in vector *wt* modulo the size of the element in bits plus 1.

The operands and results are values in integer data format *df*.

**Restrictions:**

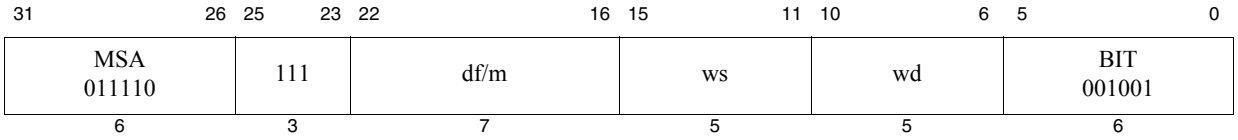
No data-dependent exceptions are possible.

**Operation:**

```
BINSR.B
    for i in 0 .. WRLEN/8-1
        t ← WR[wt]8i+2..8i
        WR[wd]8i+7..8i ← WR[wd]8i+7..8i+t+1 || WR[ws]8i+t..8i
    endfor

BINSR.H
    for i in 0 .. WRLEN/16-1
        t ← WR[wt]16i+3..16i
        WR[wd]16i+15..16i ← WR[wd]16i+15..16i+t+1 || WR[ws]16i+t..16i
    endfor

BINSR.W
    for i in 0 .. WRLEN/32-1
        t ← WR[wt]32i+4..32i
        WR[wd]32i+31..32i ← WR[wd]32i+31..32i+t+1 || WR[ws]32i+t..32i
    endfor

BINSR.D
    for i in 0 .. WRLEN/64-1
        t ← WR[wt]64i+5..64i
        WR[wd]64i+63..64i ← WR[wd]64i+63..64i+t+1 || WR[ws]64i+t..64i
    endfor
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception.




**Format:** `BINSRI.df`

`BINSRI.B wd,ws,m` **MSA**

`BINSRI.H wd,ws,m` **MSA**

`BINSRI.W wd,ws,m` **MSA**

`BINSRI.D wd,ws,m` **MSA**

**Purpose:** Immediate Bit Insert Right

Immediate selected right most bits copy while preserving destination left bits.

**Description:** `wd[i] ← bit_insert_right(wd[i], ws[i], m)`

Copy least significant (right) bits in each element of vector *ws* to elements in vector *wd* while preserving the most significant (left) bits. The number of bits to copy is given by the immediate *m* modulo the size of the element in bits plus 1.

The operands and results are values in integer data format *df*.

**Restrictions:**

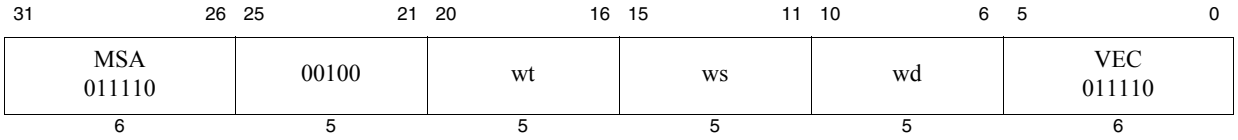
No data-dependent exceptions are possible.

**Operation:**

```
BINSRI.B
    t ← m
    for i in 0 .. WRLEN/8-1
        WR[wd]8i+7..8i ← WR[wd]8i+7..8i+7+t+1 || WR[ws]8i+t..8i
    endfor

BINSRI.H
    t ← m
    for i in 0 .. WRLEN/16-1
        WR[wd]16i+15..16i ← WR[wd]16i+15..16i+t+1 || WR[ws]16i+t..16i
    endfor

BINSRI.W
    t ← m
    for i in 0 .. WRLEN/32-1
        WR[wd]32i+31..32i ← WR[wd]32i+31..32i+t+1 || WR[ws]32i+t..32i
    endfor

BINSRI.D
    t ← m
    for i in 0 .. WRLEN/64-1
        WR[wd]64i+63..64i ← WR[wd]64i+63..64i+t+1 || WR[ws]64i+t..64i
    endfor
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception.




**Format:** `BMNZ.V`
`BMNZ.V wd,ws,wt` **MSA**

**Purpose:** Vector Bit Move If Not Zero

Vector mask-based copy bits on the condition mask being set.

**Description:** `wd ← (ws AND wt) OR (wd AND NOT wt)`

Copy to destination vector *wd* all bits from source v ector *ws* for which the corresponding bits from target vector *wt* are 1 and leaves unchanged all destination bits for which the corresponding target bits are 0.

The operands and results are bit vector values.

**Restrictions:**

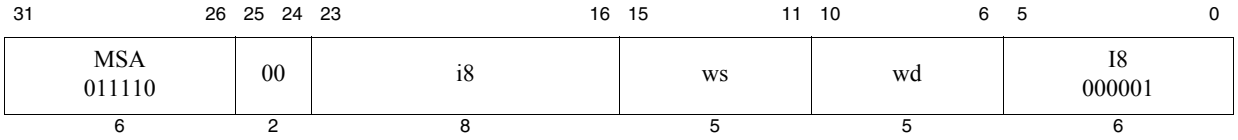
No data-dependent exceptions are possible.

**Operation:**

```
WR[wd] ← (WR[ws] and WR[wt]) or (WR[wd] and not WR[wt])
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception.




**Format:** `BMNZI.B`
`BMNZI.B wd,ws,i8` **MSA**

**Purpose:** Immediate Bit Move If Not Zero

Immediate mask-based copy bits on the condition mask being set.

**Description:** `wd[i] ← (ws[i] AND i8) OR (wd[i] AND NOT i8)`

Copy to destination vector *wd* all bits from source vector *ws* for which the corresponding bits from immediate *i8* are 1 and leaves unchanged all destination bits for which the corresponding immediate bits are 0.

The operands and results are vector values in integer byte data format.

**Restrictions:**

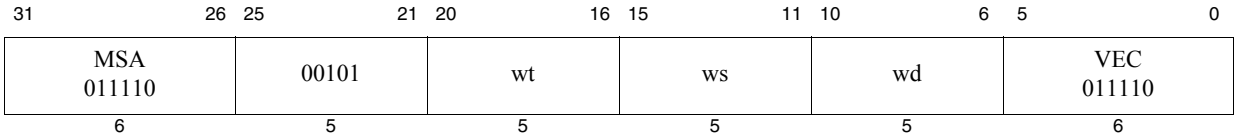
No data-dependent exceptions are possible.

**Operation:**

```
WR[wd] ← (WR[ws]8i+7..8i and i87..0) or (WR[wd]8i+7..8i and not i87..0)
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception.




**Format:** `BMZ.V`
`BMZ.V wd,ws,wt` **MSA**

**Purpose:** Vector Bit Move If Zero

Vector mask-based copy bits on the condition mask being clear.

**Description:** `wd ← (ws AND NOT wt) OR (wd AND wt)`

Copy to destination vector *wd* all bits from source v ector *ws* for which the corresponding bits from target vector *wt* are 0 and leaves unchanged all destination bits for which the corresponding target bits are 1.

The operands and results are bit vector values.

**Restrictions:**

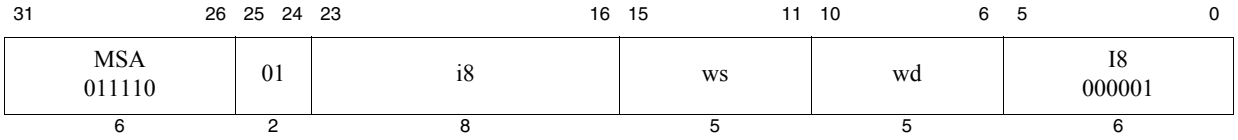
No data-dependent exceptions are possible.

**Operation:**

```
WR[wd] ← (WR[ws] and not WR[wt]) or (WR[wd] and WR[wt])
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception.




**Format:** `BMZI.B`
`BMZI.B wd,ws,i8` **MSA**

**Purpose:** Immediate Bit Move If Zero

Immediate mask-based copy bits on the condition mask being clear.

**Description:** `wd[i] ← (ws[i] AND NOT i8) OR (wd[i] AND i8)`

Copy to destination vector *wd* all bits from source vector *ws* for which the corresponding bits from immediate *i8* are 0 and leaves unchanged all destination bits for which the corresponding immediate bits are 1.

The operands and results are vector values in integer byte data format.

**Restrictions:**

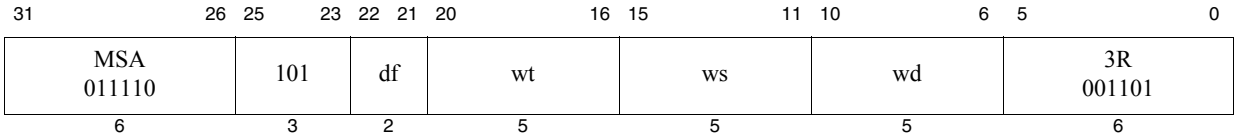
No data-dependent exceptions are possible.

**Operation:**

```
WR[wd] ← (WR[ws] and not i8_7..0) or (WR[wd] and i8_7..0)
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception.




**Format:** `BNEG.df`

`BNEG.B wd,ws,wt` **MSA**

`BNEG.H wd,ws,wt` **MSA**

`BNEG.W wd,ws,wt` **MSA**

`BNEG.D wd,ws,wt` **MSA**

**Purpose:** Vector Bit Negate

Vector selected bit position negate in each element.

**Description:** `wd[i] ← bit_negate(ws[i], wt[i])`

Negate (complement) one bit in each element of vector *ws*. The bit position is given by the elements in *wt* modulo the size of the element in bits. The result is written to vector *wd*.

The operands and results are values in integer data format *df*.

**Restrictions:**

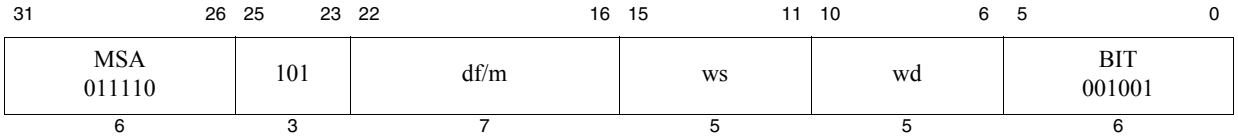
No data-dependent exceptions are possible.

**Operation:**

```
BNEG.B
    for i in 0 .. WRLEN/8-1
        t ← WR[wt]8i+2..8i
        WR[wd]8i+7..8i ← WR[ws]8i+7..8i xor (0^(7-t) || 1 || 0^t)
    endfor

BNEG.H
    for i in 0 .. WRLEN/16-1
        t ← WR[wt]16i+3..16i
        WR[wd]16i+15..16i ← WR[ws]16i+15..16i xor (0^(15-t) || 1 || 0^t)
    endfor

BNEG.W
    for i in 0 .. WRLEN/32-1
        t ← WR[wt]32i+4..32i
        WR[wd]32i+31..32i ← WR[ws]32i+31..32i xor (0^(31-t) || 1 || 0^t)
    endfor

BNEG.D
    for i in 0 .. WRLEN/64-1
        t ← WR[wt]64i+5..64i
        WR[wd]64i+63..64i ← WR[ws]64i+63..64i xor (0^(63-t) || 1 || 0^t)
    endfor
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception.




**Format:** `BNEGI.df`

`BNEGI.B wd,ws,m` **MSA**

`BNEGI.H wd,ws,m` **MSA**

`BNEGI.W wd,ws,m` **MSA**

`BNEGI.D wd,ws,m` **MSA**

**Purpose:** Immediate Bit Negate

Immediate selected bit position negate in each element.

**Description:** `wd[i] ← bit_negate(ws[i], m)`

Negate (complement) one bit in each element of vector *ws*. The bit position is given by the immediate *m* modulo the size of the element in bits. The result is written to vector *wd*.

The operands and results are values in integer data format *df*.

**Restrictions:**

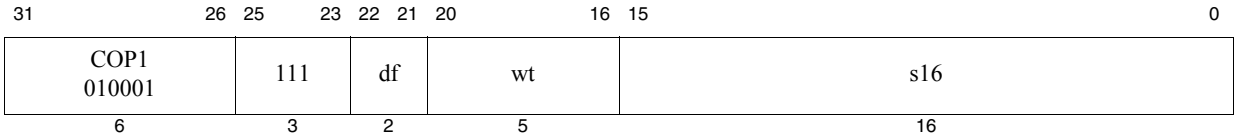
No data-dependent exceptions are possible.

**Operation:**

```
BNEGI.B
   t ← m
   for i in 0 .. WRLEN/8-1
      WR[wd]8i+7..8i ← WR[ws]8i+7..8i xor (0^(7-t) || 1 || 0^t)
   endfor

BNEGI.H
   t ← m
   for i in 0 .. WRLEN/16-1
      WR[wd]16i+15..16i ← WR[ws]16i+15..16i xor (0^(15-t) || 1 || 0^t)
   endfor

BNEGI.W
   t ← m
   for i in 0 .. WRLEN/32-1
      WR[wd]32i+31..32i ← WR[ws]32i+31..32i xor (0^(31-t) || 1 || 0^t)
   endfor

BNEGI.D
   t ← m
   for i in 0 .. WRLEN/64-1
      WR[wd]64i+63..64i ← WR[ws]64i+63..64i xor (0^(63-t) || 1 || 0^t)
   endfor
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception.




**Format:** `BNZ.df`

`BNZ.B wt,s16` **MSA**

`BNZ.H wt,s16` **MSA**

`BNZ.W wt,s16` **MSA**

`BNZ.D wt,s16` **MSA**

**Purpose:** Immediate Branch If All Elements Are Not Zero

Immediate PC offset branch if all destination elements are not zero.

**Description:** `if wt[i] ≠ 0 for all i then branch PC-relative s16`

PC-relative branch if all elements in *wt* are not zero.

The branch instruction has a delay slot. *s16* is a PC word offset, i.e. signed count of 32-bit instructions, from the PC of the delay slot.

**Restrictions:**

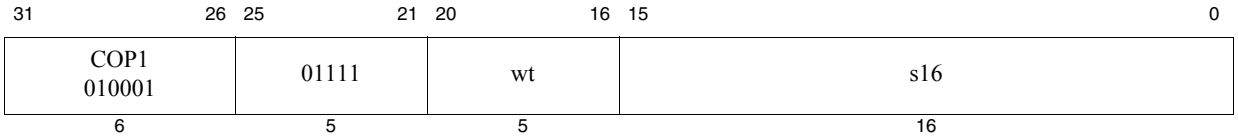
Processor operation is **UNPREDICTABLE** if a branch is placed in the delay slot of a branch or jump.

**Operation:**

```
BNZ.B
   branch(WR[wt]8i+7..8i ≠ 0 for all i, s16)

BNZ.H
   branch(WR[wt]16i+15..16i ≠ 0 for all i, s16)

BNZ.W
   branch(WR[wt]32i+31..32i ≠ 0 for all i, s16)

BNZ.D
   branch(WR[wt]64i+63..64i ≠ 0 for all i, s16)

function branch(cond, offset)
   if cond then
      I:   target_offset ← (offset9)GPRLEN-12 || offset9..0 || 0^^2
      I+1: PC ← PC + target_offset
   endif
endfunction branch
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception.




**Format:** `BNZ.V`
`BNZ.V wt,s16` **MSA**

**Purpose:** Immediate Branch If Not Zero (At Least One Element of Any Format Is Not Zero)

Immediate PC offset branch if destination vector is not zero.

**Description:** `if wt ≠ 0 then branch PC-relative s16`

PC-relative branch if at least one bit in *wt* is not zero, i.e at least one element is not zero regardless of the data format.

The branch instruction has a delay slot. *s16* is a PC word offset, i.e. signed count of 32-bit instructions, from the PC of the delay slot.

**Restrictions:**

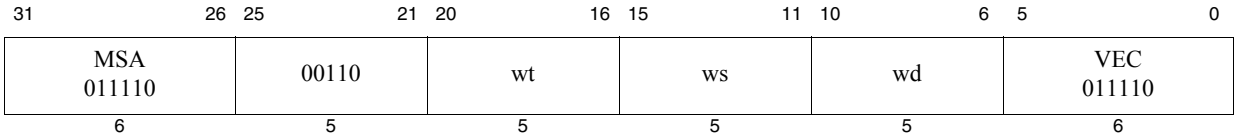
Processor operation is **UNPREDICTABLE** if a branch is placed in the delay slot of a branch or jump.

**Operation:**

```
    branch(WR[wt] ≠ 0, s16)

function branch(cond, offset)
   if cond then
      I:   target_offset ← (offset9)GPRLEN-12 || offset9..0 || 0^^2
      I+1: PC ← PC + target_offset
   endif
endfunction branch
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception.




**Format:** `BSEL.V`
`BSEL.V wd,ws,wt` **MSA**

**Purpose:** Vector Bit Select

Vector mask-based copy bits from two source vectors selected by the bit mask value

**Description:** `wd ← (ws AND NOT wd) OR (wt AND wd)`

Selectively copy bits from the source v ectors *ws* and *wt* into destination vector *wd* based on the corresponding bit in *wd*: if 0 copies the bit from *ws*, if 1 copies the bit from *wt*.

**Restrictions:**

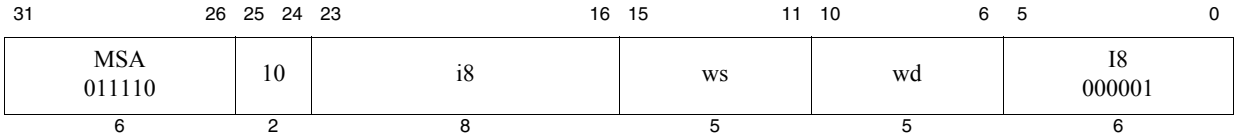
The operands and results are bit vector values.

**Operation:**

```
WR[wd] ← (WR[ws] and not WR[wd]) or (WR[wt] and WR[wd])
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception.




**Format:** `BSELI.B`
`BSELI.B wd,ws,i8` **MSA**

**Purpose:** Immediate Bit Select

Immediate mask-based copy bits from two source vectors selected by the bit mask value

**Description:** `wd ← (ws AND NOT wd) OR (i8 AND wd)`

Selectively copy bits from the the 8-bit immediate *i8* and source vector *ws* into destination vector *wd* based on the corresponding bit in *wd*: if 0 copies the bit from *ws*, if 1 copies the bit from *i8*.

**Restrictions:**

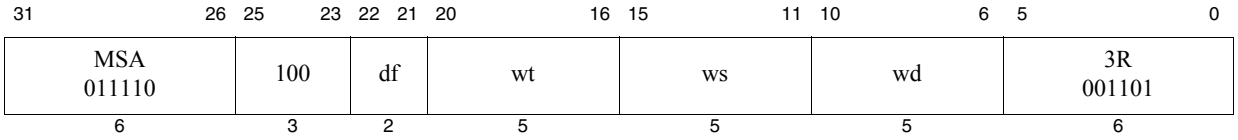
The operands and results are bit vector values.

**Operation:**

```
for i in 0 .. WRLEN/8-1
  WR[wd]8i+7..8i ←
      (WR[ws]8i+7..8i and not WR[wd]8i+7..8i) or (i87..0 and WR[wd]8i+7..8i)
endfor
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception.




**Format:** `BSET.df`

`BSET.B wd,ws,wt` **MSA**

`BSET.H wd,ws,wt` **MSA**

`BSET.W wd,ws,wt` **MSA**

`BSET.D wd,ws,wt` **MSA**

**Purpose:** Vector Bit Set

Vector selected bit position set in each element.

**Description:** `wd[i] ← bit_set(ws[i], wt[i])`

Set to 1 one bit in each element of vector *ws*. The bit position is given by the elements in *wt* modulo the size of the element in bits. The result is written to vector *wd*.

The operands and results are values in integer data format *df*.

**Restrictions:**

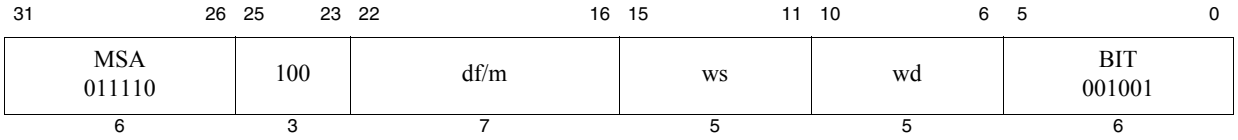
No data-dependent exceptions are possible.

**Operation:**

```
BSET_S.B
    for i in 0 .. WRLEN/8-1
        t ← WR[wt]8i+2..8i
        WR[wd]8i+7..8i ← WR[ws]8i+7..8i or (0^(7-t) || 1 || 0^t)
    endfor

BSET_S.H
    for i in 0 .. WRLEN/16-1
        t ← WR[wt]16i+3..16i
        WR[wd]16i+15..16i ← WR[ws]16i+15..16i or (0^(15-t) || 1 || 0^t)
    endfor

BSET_S.W
    for i in 0 .. WRLEN/32-1
        t ← WR[wt]32i+4..32i
        WR[wd]32i+31..32i ← WR[ws]32i+31..32i or (0^(31-t) || 1 || 0^t)
    endfor

BSET_S.D
    for i in 0 .. WRLEN/64-1
        t ← WR[wt]64i+5..64i
        WR[wd]64i+63..64i ← WR[ws]64i+63..64i or (0^(63-t) || 1 || 0^t)
    endfor
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception.




**Format:** `BSETI.df`

`BSETI.B wd,ws,m` **MSA**

`BSETI.H wd,ws,m` **MSA**

`BSETI.W wd,ws,m` **MSA**

`BSETI.D wd,ws,m` **MSA**

**Purpose:** Immediate Bit Set

Immediate selected bit position set in each element.

**Description:** `wd[i] ← bit_set(ws[i], m)`

Set to 1 one bit in each element of vector *ws*. The bit position is given by the immediate *m*. The result is written to vector *wd*.

The operands and results are values in integer data format *df*.

**Restrictions:**

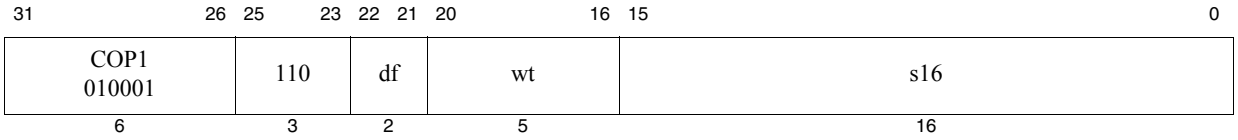
No data-dependent exceptions are possible.

**Operation:**

```
BSETI_S.B
   t ← m
   for i in 0 .. WRLEN/8-1
      WR[wd]8i+7..8i ← WR[ws]8i+7..8i or (0^(7-t) || 1 || 0^t)
   endfor

BSETI_S.H
   t ← m
   for i in 0 .. WRLEN/16-1
      WR[wd]16i+15..16i ← WR[ws]16i+15..16i or (0^(15-t) || 1 || 0^t)
   endfor

BSETI_S.W
   t ← m
   for i in 0 .. WRLEN/32-1
      WR[wd]32i+31..32i ← WR[ws]32i+31..32i or (0^(31-t) || 1 || 0^t)
   endfor

BSETI_S.D
   t ← m
   for i in 0 .. WRLEN/64-1
      WR[wd]64i+63..64i ← WR[ws]64i+63..64i or (0^(63-t) || 1 || 0^t)
   endfor
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception.




**Format:** `BZ.df`

`BZ.B wt,s16` **MSA**

`BZ.H wt,s16` **MSA**

`BZ.W wt,s16` **MSA**

`BZ.D wt,s16` **MSA**

**Purpose:** Immediate Branch If At Least One Element Is Zero

Immediate PC offset branch if at least one destination element is zero.

**Description:** `if wt[i] = 0 for some i then branch PC-relative s16`

PC-relative branch if at least one element in *wt* is zero.

The branch instruction has a delay slot. *s16* is a PC word offset, i.e. signed count of 32-bit instructions, from the PC of the delay slot.

**Restrictions:**

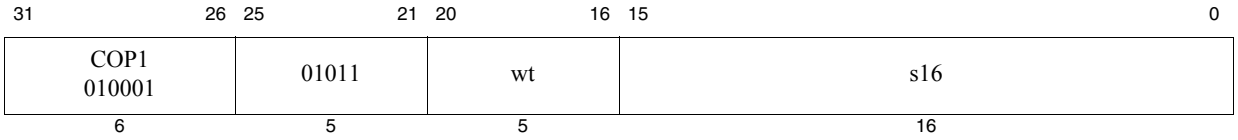
Processor operation is **UNPREDICTABLE** if a branch is placed in the delay slot of a branch or jump.

**Operation:**

```
BZ.B
   for i in 0 .. WRLEN/8-1
      branch(WR[wt]8i+7..8i = 0, s16)
   endfor

BZ.H
   for i in 0 .. WRLEN/16-1
      branch(WR[wt]16i+15..16i = 0, s16)
   endfor

BZ.W
   for i in 0 .. WRLEN/32-1
      branch(WR[wt]32i+31..32i = 0, s16)
   endfor

BZ.D
   for i in 0 .. WRLEN/64-1
      branch(WR[wt]64i+63..64i = 0, s16)
   endfor

function branch(cond, offset)
   if cond then
      I:   target_offset ← (offset9)GPRLEN-12 || offset9..0 || 0^^2
      I+1: PC ← PC + target_offset
   endif
endfunction branch
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception.




**Format:** `BZ.V`
`BZ.V wt,s16` **MSA**

**Purpose:** Immediate Branch If Zero (All Elements of Any Format Are Zero)

Immediate PC offset branch if destination vector is zero.

**Description:** `if wt = 0 then branch PC-relative s16`

PC-relative branch if all *wt* bits are zero, i.e. all elements are zero regardless of the data format.

The branch instruction has a delay slot. *s16* is a PC word offset, i.e. signed count of 32-bit instructions, from the PC of the delay slot.

**Restrictions:**

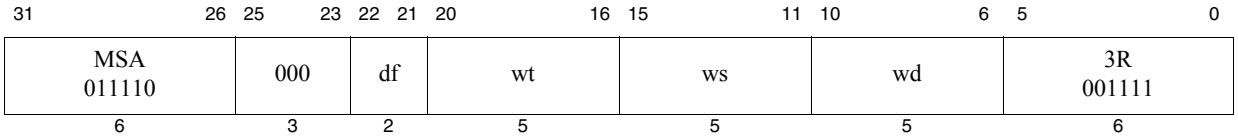
Processor operation is **UNPREDICTABLE** if a branch is placed in the delay slot of a branch or jump.

**Operation:**

```
    branch(WR[wt] = 0, s16)

function branch(cond, offset)
   if cond then
      I:   target_offset ← (offset9)GPRLEN-12 || offset9..0 || 0^^2
      I+1: PC ← PC + target_offset
   endif
endfunction branch
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception.




**Format:** `CEQ.df`

`CEQ.B wd,ws,wt` **MSA**

`CEQ.H wd,ws,wt` **MSA**

`CEQ.W wd,ws,wt` **MSA**

`CEQ.D wd,ws,wt` **MSA**

**Purpose:** Vector Compare Equal

Vector to vector compare for equality; if true all destination bits are set, otherwise clear.

**Description:** `wd[i] ← (ws[i] = wt[i])`

Set all bits to 1 in *wd* elements if the corresponding *ws* and *wt* elements are equal, otherwise set all bits to 0.

The operands and results are values in integer data format *df*.

**Restrictions:**

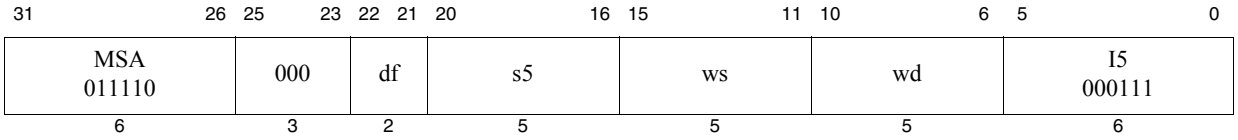
No data-dependent exceptions are possible.

**Operation:**

```
CEQ.B
    for i in 0 .. WRLEN/8-1
        c ← WR[ws]8i+7..8i = WR[wt]8i+7..8i
        WR[wd]8i+7..8i ← c8
    endfor

CEQ.H
    for i in 0 .. WRLEN/16-1
        c ← WR[ws]16i+15..16i = WR[wt]16i+15..16i
        WR[wd]16i+15..16i ← c16
    endfor

CEQ.W
    for i in 0 .. WRLEN/32-1
        c ← WR[ws]32i+31..32i = WR[wt]32i+31..32i
        WR[wd]32i+31..32i ← c32
    endfor

CEQ.D
    for i in 0 .. WRLEN/64-1
        c ← WR[ws]64i+63..64i = WR[wt]64i+63..64i
        WR[wd]64i+63..64i ← c64
    endfor
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception.




**Format:** `CEQI.df`

`CEQI.B wd,ws,s5` **MSA**

`CEQI.H wd,ws,s5` **MSA**

`CEQI.W wd,ws,s5` **MSA**

`CEQI.D wd,ws,s5` **MSA**

**Purpose:** Immediate Compare Equal

Immediate to vector compare for equality; if true all destination bits are set, otherwise clear.

**Description:** `wd[i] ← (ws[i] = s5)`

Set all bits to 1 in *wd* elements if the corresponding *ws* element and the 5-bit signed immediate *s5* are equal, otherwise set all bits to 0.

The operands and results are values in integer data format *df*.

**Restrictions:**

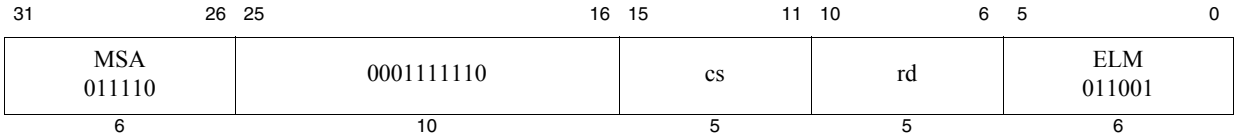
No data-dependent exceptions are possible.

**Operation:**

```
CEQI.B
    t ← (s5_4)^3 || s5_4..0
    for i in 0 .. WRLEN/8-1
        c ← WR[ws]_8i+7..8i = t
        WR[wd]_8i+7..8i ← c^8
    endfor

CEQI.H
    t ← (s5_4)^11 || s5_4..0
    for i in 0 .. WRLEN/16-1
        c ← WR[ws]_16i+15..16i = t
        WR[wd]_16i+15..16i ← c^16
    endfor

CEQI.W
    t ← (s5_4)^27 || s5_4..0
    for i in 0 .. WRLEN/32-1
        c ← WR[ws]_32i+31..32i = t
        WR[wd]_32i+31..32i ← c^32
    endfor

CEQI.D
    t ← (s5_4)^59 || s5_4..0
    for i in 0 .. WRLEN/64-1
        c ← WR[ws]_64i+63..64i = t
        WR[wd]_64i+63..64i ← c^64
    endfor
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception.




**Format:** `CFCMSA`
`CFCMSA rd,cs` **MSA**

**Purpose:** GPR Copy from MSA Control Register

GPR value copied from MSA control register.

**Description:** `rd ← signed(cs)`

The sign extended content of MSA control register *cs* is copied to GPR *rd*.

**Restrictions:**

The read operation returns ZERO if *cs* specifies a reserved register or a register that does not exist.

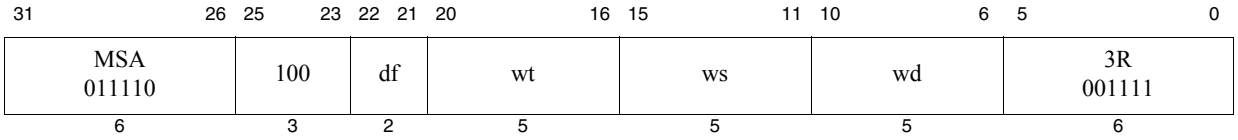
**Operation:**

```
if cs = 0 then
   GPR[rd] ← sign_extend(MSAIR, 64)
elseif cs = 1 then
   GPR[rd] ← sign_extend(MSACSR, 64)
elseif MSAIR_WRP = 1 then
   if cs = 2 then
      if not IsCoprocessorEnabled(0) then
         SignalException(CoprocessorUnusableException, 0)
      endif
      GPR[rd] ← sign_extend(MSAAccess, 64)
   elseif cs = 3 then
      if not IsCoprocessorEnabled(0) then
         SignalException(CoprocessorUnusableException, 0)
      endif
      GPR[rd] ← sign_extend(MSASave, 64)
   elseif cs = 4 then
      if not IsCoprocessorEnabled(0) then
         SignalException(CoprocessorUnusableException, 0)
      endif
      GPR[rd] ← sign_extend(MSAModify, 64)
   elseif cs = 5 then
      if not IsCoprocessorEnabled(0) then
         SignalException(CoprocessorUnusableException, 0)
      endif
      GPR[rd] ← sign_extend(MSARequest, 64)
   elseif cs = 6 then
      if not IsCoprocessorEnabled(0) then
         SignalException(CoprocessorUnusableException, 0)
      endif
      GPR[rd] ← sign_extend(MSAMap, 64)
   elseif cs = 7 then
      if not IsCoprocessorEnabled(0) then
         SignalException(CoprocessorUnusableException, 0)
      endif
      GPR[rd] ← sign_extend(MSAUnmap, 64)
   else
```

```
        GPR[rd] = 0
    endif
else
    GPR[rd] = 0
endif
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception. Coprocessor 0 Unusable Exception.




**Format:** `CLE_S.df`

`CLE_S.B wd,ws,wt` **MSA**

`CLE_S.H wd,ws,wt` **MSA**

`CLE_S.W wd,ws,wt` **MSA**

`CLE_S.D wd,ws,wt` **MSA**

**Purpose:** Vector Compare Signed Less Than or Equal

Vector to vector compare for signed less or equal; if true all destination bits are set, otherwise clear.

**Description:** `wd[i] ← (ws[i] <= wt[i])`

Set all bits to 1 in *wd* elements if the corresponding *ws* elements are signed less than or equal to *wt* elements, otherwise set all bits to 0.

The operands and results are values in integer data format *df*.

**Restrictions:**

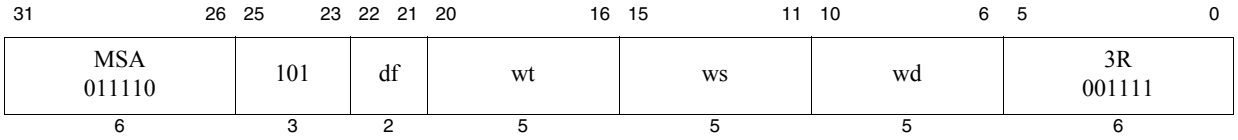
No data-dependent exceptions are possible.

**Operation:**

```
CLE_S.B
    for i in 0 .. WRLEN/8-1
        c ← WR[ws]8i+7..8i <= WR[wt]8i+7..8i
        WR[wd]8i+7..8i ← c8
    endfor

CLE_S.H
    for i in 0 .. WRLEN/16-1
        c ← WR[ws]16i+15..16i <= WR[wt]16i+15..16i
        WR[wd]16i+15..16i ← c16
    endfor

CLE_S.W
    for i in 0 .. WRLEN/32-1
        c ← WR[ws]32i+31..32i <= WR[wt]32i+31..32i
        WR[wd]32i+31..32i ← c32
    endfor

CLE_S.D
    for i in 0 .. WRLEN/64-1
        c ← WR[ws]64i+63..64i <= WR[wt]64i+63..64i
        WR[wd]64i+63..64i ← c64
    endfor
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception.




**Format:** `CLE_U.df`

`CLE_U.B wd,ws,wt` **MSA**

`CLE_U.H wd,ws,wt` **MSA**

`CLE_U.W wd,ws,wt` **MSA**

`CLE_U.D wd,ws,wt` **MSA**

**Purpose:** Vector Compare Unsigned Less Than or Equal

Vector to vector compare for unsigned less or equal; if true all destination bits are set, otherwise clear.

**Description:** `wd[i] ← (ws[i] <= wt[i])`

Set all bits to 1 in *wd* elements if the corresponding *ws* elements are unsigned less than or equal to *wt* elements, otherwise set all bits to 0.

The operands and results are values in integer data format *df*.

**Restrictions:**

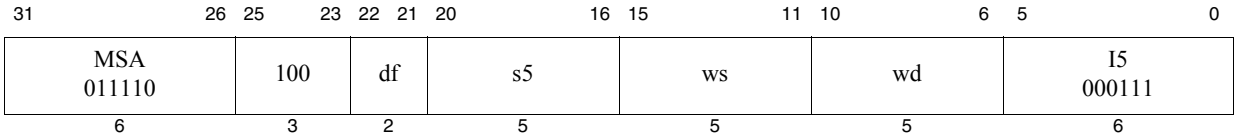
No data-dependent exceptions are possible.

**Operation:**

```
CLE_U.B
    for i in 0 .. WRLEN/8-1
        c ← (0 || WR[ws]8i+7..8i) <= (0 || WR[wt]8i+7..8i)
        WR[wd]8i+7..8i ← c8
    endfor

CLE_U.H
    for i in 0 .. WRLEN/16-1
        c ← (0 || WR[ws]16i+15..16i) <= (0 || WR[wt]16i+15..16i)
        WR[wd]16i+15..16i ← c16
    endfor

CLE_U.W
    for i in 0 .. WRLEN/32-1
        c ← (0 || WR[ws]__32i+31..32i_)_ <= (0 || WR[wt]32i+31..32i)
        WR[wd]32i+31..32i ← c32
    endfor

CLE_U.D
    for i in 0 .. WRLEN/64-1
        c ← (0 || WR[ws]64i+63..64i) <= (0 || WR[wt]64i+63..64i)
        WR[wd]64i+63..64i ← c64
    endfor
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception.




**Format:** `CLEI_S.df`

`CLEI_S.B wd,ws,s5` **MSA**

`CLEI_S.H wd,ws,s5` **MSA**

`CLEI_S.W wd,ws,s5` **MSA**

`CLEI_S.D wd,ws,s5` **MSA**

**Purpose:** Immediate Compare Signed Less Than or Equal

Immediate to vector compare for signed less or equal; if true all destination bits are set, otherwise clear.

**Description:** `wd[i] ← (ws[i] <= s5)`

Set all bits to 1 in *wd* elements if the corresponding *ws* element is less than or equal to the 5-bit signed immediate *s5*, otherwise set all bits to 0.

The operands and results are values in integer data format *df*.

**Restrictions:**

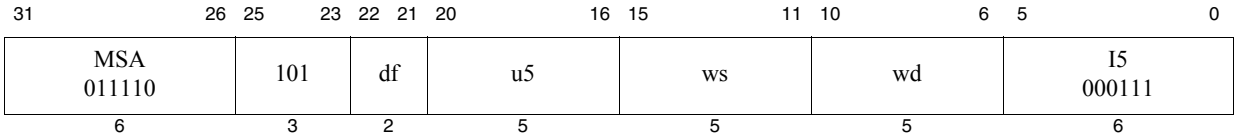
No data-dependent exceptions are possible.

**Operation:**

```
CLEI_S.B
    t ← (s5_4)^3 || s5_4..0
    for i in 0 .. WRLEN/8-1
        c ← WR[ws]_8i+7..8i <= t
        WR[wd]_8i+7..8i ← c^8
    endfor

CLEI_S.H
    t ← (s5_4)^11 || s5_4..0
    for i in 0 .. WRLEN/16-1
        c ← WR[ws]_16i+15..16i <= t
        WR[wd]_16i+15..16i ← c^16
    endfor

CLEI_S.W
    t ← (s5_4)^27 || s5_4..0
    for i in 0 .. WRLEN/32-1
        c ← WR[ws]_32i+31..32i <= t
        WR[wd]_32i+31..32i ← c^32
    endfor

CLEI_S.D
    t ← (s5_4)^59 || s5__4.. 0__
    for i in 0 .. WRLEN/64-1
        c ← WR[ws]_64i+63..64i <= t
        WR[wd]_64i+63..64i ← c^64
    endfor
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception.




**Format:** `CLEI_U.df`

`CLEI_U.B wd,ws,u5` **MSA**

`CLEI_U.H wd,ws,u5` **MSA**

`CLEI_U.W wd,ws,u5` **MSA**

`CLEI_U.D wd,ws,u5` **MSA**

**Purpose:** Immediate Compare Unsigned Less Than or Equal

Immediate to vector compare for unsigned less or equal; if true all destination bits are set, otherwise clear.

**Description:** `wd[i] ← (ws[i] <= u5)`

Set all bits to 1 in *wd* elements if the corresponding *ws* element is unsigned less than or equal to the 5-bit unsigned immediate *u5*, otherwise set all bits to 0.

The operands and results are values in integer data format *df*.

**Restrictions:**

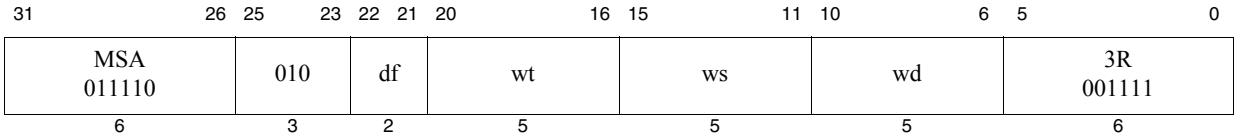
No data-dependent exceptions are possible.

**Operation:**

```
CLEI_U.B
    t ← 0^3 || u5_4..0
    for i in 0 .. WRLEN/8-1
        c ← (0 || WR[ws]_8i+7..8i) <= (0 || t)
        WR[wd]_8i+7..8i ← c^8
    endfor

CLEI_U.H
    t ← 0^11 || u5_4..0
    for i in 0 .. WRLEN/16-1
        c ← (0 || WR[ws]_16i+15..16i) <= (0 || t)
        WR[wd]_16i+15..16i ← c^16
    endfor

CLEI_U.W
    t ← 0^27 || u5_4..0
    for i in 0 .. WRLEN/32-1
        c ← WR[ws]_32i+31..32i <= (0 || t)
        WR[wd]_32i+31..32i ← c^32
    endfor

CLEI_U.D
    t ← 0^59 || u5_4..0
    for i in 0 .. WRLEN/64-1
        c ← WR[ws]_64i+63..64i <= (0 || t)
        WR[wd]_64i+63..64i ← c^64
    endfor
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception.




**Format:** `CLT_S.df`

`CLT_S.B wd,ws,wt` **MSA**

`CLT_S.H wd,ws,wt` **MSA**

`CLT_S.W wd,ws,wt` **MSA**

`CLT_S.D wd,ws,wt` **MSA**

**Purpose:** Vector Compare Signed Less Than

Vector to vector compare for signed less than; if true all destination bits are set, otherwise clear.

**Description:** `wd[i] ← (ws[i] < wt[i])`

Set all bits to 1 in *wd* elements if the corresponding *ws* elements are signed less than *wt* elements, otherwise set all bits to 0.

The operands and results are values in integer data format *df*.

**Restrictions:**

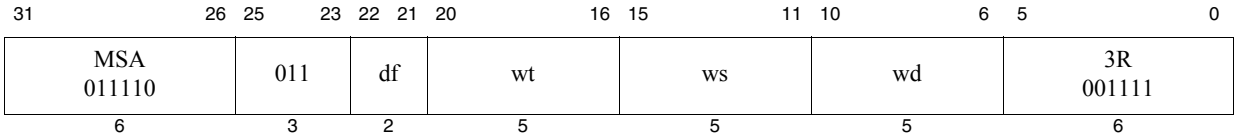
No data-dependent exceptions are possible.

**Operation:**

```
CLT_S.B
    for i in 0 .. WRLEN/8-1
        c ← WR[ws]8i+7..8i < WR[wt]8i+7..8i
        WR[wd]8i+7..8i ← c8
    endfor

CLT_S.H
    for i in 0 .. WRLEN/16-1
        c ← WR[ws]16i+15..16i < WR[wt]16i+15..16i
        WR[wd]16i+15..16i ← c16
    endfor

CLT_S.W
    for i in 0 .. WRLEN/32-1
        c ← WR[ws]32i+31..32i < WR[wt]32i+31..32i
        WR[wd]32i+31..32i ← c32
    endfor

CLT_S.D
    for i in 0 .. WRLEN/64-1
        c ← WR[ws]64i+63..64i < WR[wt]64i+63..64i
        WR[wd]64i+63..64i ← c64
    endfor
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception.




**Format:** `CLT_U.df`

`CLT_U.B wd,ws,wt` **MSA**

`CLT_U.H wd,ws,wt` **MSA**

`CLT_U.W wd,ws,wt` **MSA**

`CLT_U.D wd,ws,wt` **MSA**

**Purpose:** Vector Compare Unsigned Less Than

Vector to vector compare for unsigned less than; if true all destination bits are set, otherwise clear.

**Description:** `wd[i] ← (ws[i] < wt[i])`

Set all bits to 1 in *wd* elements if the corresponding *ws* elements are unsigned less than *wt* elements, otherwise set all bits to 0.

The operands and results are values in integer data format *df*.

**Restrictions:**

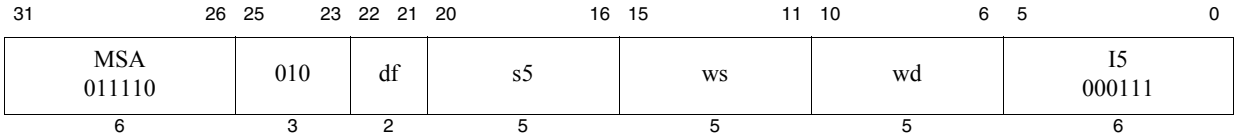
No data-dependent exceptions are possible.

**Operation:**

```
CLT_U.B
    for i in 0 .. WRLEN/8-1
        c ← (0 || WR[ws]8i+7..8i) < (0 || WR[wt]8i+7..8i)
        WR[wd]8i+7..8i ← c8
    endfor

CLT_U.H
    for i in 0 .. WRLEN/16-1
        c ← (0 || WR[ws]16i+15..16i) < (0 || WR[wt]16i+15..16i)
        WR[wd]16i+15..16i ← c16
    endfor

CLT_U.W
    for i in 0 .. WRLEN/32-1
        c ← (0 || WR[ws]__32i+31..32i_)_ < (0 || WR[wt]32i+31..32i)
        WR[wd]32i+31..32i ← c32
    endfor

CLT_U.D
    for i in 0 .. WRLEN/64-1
        c ← (0 || WR[ws]64i+63..64i) < (0 || WR[wt]64i+63..64i)
        WR[wd]64i+63..64i ← c64
    endfor
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception.




**Format:** `CLTI_S.df`

`CLTI_S.B wd,ws,s5` **MSA**

`CLTI_S.H wd,ws,s5` **MSA**

`CLTI_S.W wd,ws,s5` **MSA**

`CLTI_S.D wd,ws,s5` **MSA**

**Purpose:** Immediate Compare Signed Less Than

Immediate to vector compare for signed less than; if true all destination bits are set, otherwise clear.

**Description:** `wd[i] ← (ws[i] < s5)`

Set all bits to 1 in *wd* elements if the corresponding *ws* element is less than the 5-bit signed immediate *s5*, otherwise set all bits to 0.

The operands and results are values in integer data format *df*.

**Restrictions:**

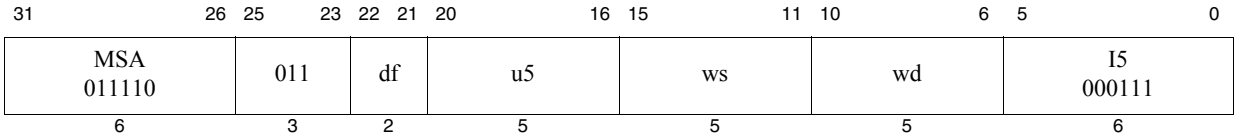
No data-dependent exceptions are possible.

**Operation:**

```
CLTI_S.B
    t ← (s5_4)^3 || s5_4..0
    for i in 0 .. WRLEN/8-1
        c ← WR[ws]_8i+7..8i < t
        WR[wd]_8i+7..8i ← c^8
    endfor

CLTI_S.H
    t ← (s5_4)^11 || s5_4..0
    for i in 0 .. WRLEN/16-1
        c ← WR[ws]_16i+15..16i < t
        WR[wd]_16i+15..16i ← c^16
    endfor

CLTI_S.W
    t ← (s5_4)^27 || s5_4..0
    for i in 0 .. WRLEN/32-1
        c ← WR[ws]_32i+31..32i < t
        WR[wd]_32i+31..32i ← c^32
    endfor

CLTI_S.D
    t ← (s5_4)^59 || s5__4.. 0__
    for i in 0 .. WRLEN/64-1
        c ← WR[ws]_64i+63..64i < t
        WR[wd]_64i+63..64i ← c^64
    endfor
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception.




**Format:** `CLTI_U.df`

`CLTI_U.B wd,ws,u5` **MSA**

`CLTI_U.H wd,ws,u5` **MSA**

`CLTI_U.W wd,ws,u5` **MSA**

`CLTI_U.D wd,ws,u5` **MSA**

**Purpose:** Immediate Compare Unsigned Less Than

Immediate to vector compare for unsigned less than; if true all destination bits are set, otherwise clear.

**Description:** `wd[i] ← (ws[i] < u5)`

Set all bits to 1 in *wd* elements if the corresponding *ws* element is unsigned less than the 5-bit unsigned immediate *u5*, otherwise set all bits to 0.

The operands and results are values in integer data format *df*.

**Restrictions:**

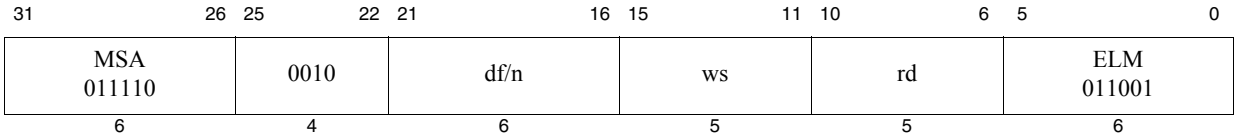
No data-dependent exceptions are possible.

**Operation:**

```
CLTI_U.B
    t ← 0^3 || u5_4..0
    for i in 0 .. WRLEN/8-1
        c ← (0 || WR[ws]_8i+7..8i) < (0 || t)
        WR[wd]_8i+7..8i ← c^8
    endfor

CLTI_U.H
    t ← 0^11 || u5_4..0
    for i in 0 .. WRLEN/16-1
        c ← (0 || WR[ws]_16i+15..16i) < (0 || t)
        WR[wd]_16i+15..16i ← c^16
    endfor

CLTI_U.W
    t ← 0^27 || u5_4..0
    for i in 0 .. WRLEN/32-1
        c ← WR[ws]_32i+31..32i < (0 || t)
        WR[wd]_32i+31..32i ← c^32
    endfor

CLTI_U.D
    t ← 0^59 || u5_4..0
    for i in 0 .. WRLEN/64-1
        c ← WR[ws]_64i+63..64i < (0 || t)
        WR[wd]_64i+63..64i ← c^64
    endfor
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception.




**Format:** `COPY_S.df`

`COPY_S.B rd,ws[n]` **MSA**

`COPY_S.H rd,ws[n]` **MSA**

`COPY_S.W rd,ws[n]` **MSA**

`COPY_S.D rd,ws[n]` **MIPS64 MSA**

**Purpose:** Element Copy to GPR Signed

Element value sign extended and copied to GPR.

**Description:** `rd ← signed(ws[n])`

Sign-extend element *n* of vector *ws* and copy the result to GPR *rd*.

**Restrictions:**

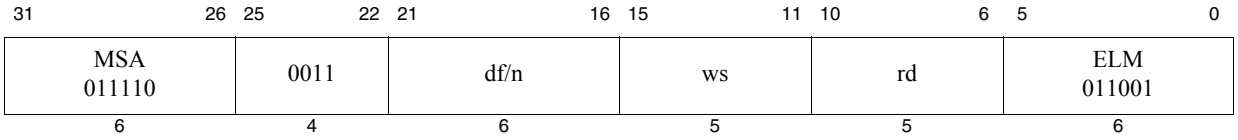
No data-dependent exceptions are possible.

**Operation:**

```
COPY_S.B
   GPR[rd] ← sign_extend(WR[ws]8n+7..8n, 64)

COPY_S.H
   GPR[rd] ← sign_extend(WR[ws]16n+15..16n, 64)

COPY_S.W
   GPR[rd] ← sign_extend(WR[ws]32n+31..32n, 64)

COPY_S.D
   GPR[rd] ← WR[ws]64n+63..64n

function sign_extend(tt, n)
   return (ttn-1)GPRLEN-n || ttn-1..0
endfunction sign_extend
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception.




**Format:** `COPY_U.df`

`COPY_U.B rd,ws[n]` **MSA**

`COPY_U.H rd,ws[n]` **MSA**

`COPY_U.W rd,ws[n]` **MIPS64 MSA**

**Purpose:** Element Copy to GPR Unsigned

Element value zero extended and copied to GPR.

**Description:** `rd ← unsigned(ws[n])`

Zero-extend element *n* of vector *ws* and copy the result to GPR *rd*.

**Restrictions:**

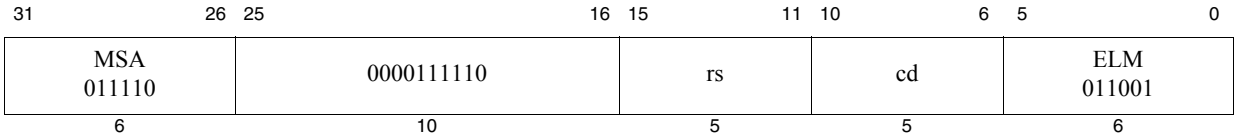
No data-dependent exceptions are possible.

**Operation:**

```
COPY_U.B
   GPR[rd] ← zero_extend(WR[ws]8n+7..8n, 64))

COPY_U.H
   GPR[rd] ← zero_extend(WR[ws]16n+15..16n, 64))

COPY_U.W
   GPR[rd] ← zero_extend(WR[ws]32n+31..32n, 64)

function zero_extend(tt, n)
   return 0^(GPRLEN-n) || tt(n-1..0)
endfunction zero_extend
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception.




**Format:** `CTCMSA`
`CTCMSA cd,rs` **MSA**

**Purpose:** GPR Copy to MSA Control Register

GPR value copied to MSA control register.

**Description:** `cd ← rs`

The content of the least significant 31 bits of GPR *rs* is copied to MSA control register *cd*.

Writing to the MSA Control and Status Register *MSACSR* causes the appropriate exception if any Cause bit and its corresponding Enable bit are both set. The register is written before the exception occurs and the EPC register contains the address of the CTCMSA instruction.

**Restrictions:**

The write attempt is IGNORED if *cd* specifies a reserved register or a register that does not exist or is not writable.

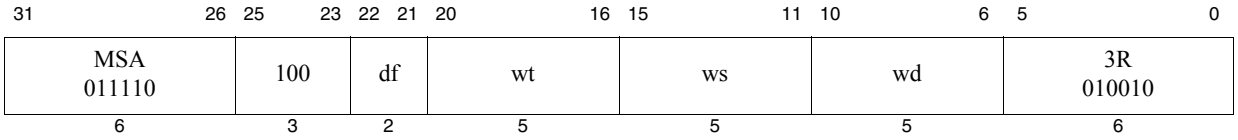
**Operation:**

```
if cd = 1 then
    MSACSR ← GPR[rs]31..0
    if MSACSRCause and (1 || MSACSREnables) ≠ 0 then
        SignalException(MSAFloatingPointException)
    endif
elseif MSAIRWRP = 1 then
    if cd = 3 then
        if not IsCoprocessorEnabled(0) then
            SignalException(CoprocessorUnusableException, 0)
        endif
        MSASave ← GPR[rs]31..0
    elseif cd = 4 then
        if not IsCoprocessorEnabled(0) then
            SignalException(CoprocessorUnusableException, 0)
        endif
        MSAModify ← GPR[rs]31..0
    elseif cd = 6 then
        if not IsCoprocessorEnabled(0) then
            SignalException(CoprocessorUnusableException, 0)
        endif
        MSAMap ← GPR[rs]31..0
    elseif cd = 7 then
        if not IsCoprocessorEnabled(0) then
            SignalException(CoprocessorUnusableException, 0)
        endif
        MSAUnmap ← GPR[rs]31..0
    endif
endif
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception, MSA Floating Point Exception. Coprocessor 0 Unusable

Exception.




**Format:** `DIV_S.df`

`DIV_S.B wd,ws,wt` **MSA**

`DIV_S.H wd,ws,wt` **MSA**

`DIV_S.W wd,ws,wt` **MSA**

`DIV_S.D wd,ws,wt` **MSA**

**Purpose:** Vector Signed Divide

Vector signed divide.

**Description:** `wd[i] ← ws[i] div wt[i]`

The signed integer elements in vector *ws* are divided by signed integer elements in vector *wt*. The result is written to vector *wd*. If a divisor element vector *wt* is zero, the result value is **UNPREDICTABLE**.

The operands and results are values in integer data format *df*.

**Restrictions:**

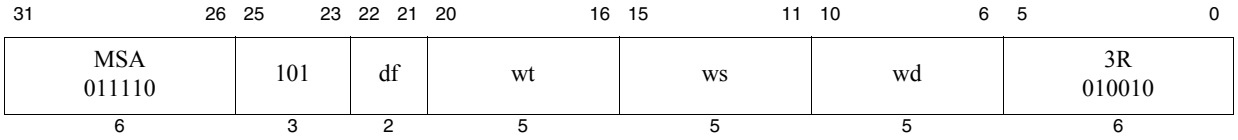
No data-dependent exceptions are possible.

**Operation:**

```
DIV_S.B
    for i in 0 .. WRLEN/8-1
           WR[wd]8i+7..8i ← WR[ws]8i+7..8i div WR[wt]8i+7..8i
    endfor

DIV_S.H
    for i in 0 .. WRLEN/16-1
        WR[wd]16i+15..16i ← WR[ws]16i+15..16i div WR[wt]16i+15..16i
    endfor

DIV_S.W
    for i in 0 .. WRLEN/32-1
        WR[wd]32i+31..32i ← WR[ws]32i+31..32i div WR[wt]32i+31..32i
    endfor

DIV_S.D
    for i in 0 .. WRLEN/64-1
        WR[wd]64i+63..64i ← WR[ws]64i+63..64i div WR[wt]64i+63..64i
    endfor
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception.




**Format:** `DIV_U.df`

`DIV_U.B wd,ws,wt` **MSA**

`DIV_U.H wd,ws,wt` **MSA**

`DIV_U.W wd,ws,wt` **MSA**

`DIV_U.D wd,ws,wt` **MSA**

**Purpose:** Vector Unsigned Divide

Vector unsigned divide.

**Description:** `wd[i] ← ws[i] udiv wt[i]`

The unsigned integer elements in vector *ws* are divided by unsigned integer elements in vector *wt*. The result is written to vector *wd*. If a divisor element vector *wt* is zero, the result value is **UNPREDICTABLE**.

The operands and results are values in integer data format *df*.

**Restrictions:**

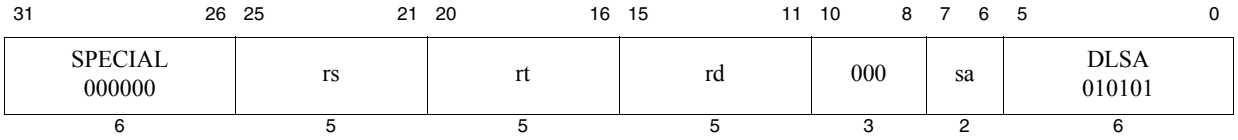
No data-dependent exceptions are possible.

**Operation:**

```
DIV_U.B
    for i in 0 .. WRLEN/8-1
           WR[wd]8i+7..8i ← WR[ws]8i+7..8i udiv WR[wt]8i+7..8i
    endfor

DIV_U.H
    for i in 0 .. WRLEN/16-1
        WR[wd]16i+15..16i ← WR[ws]16i+15..16i udiv WR[wt]16i+15..16i
    endfor

DIV_U.W
    for i in 0 .. WRLEN/32-1
        WR[wd]32i+31..32i ← WR[ws]32i+31..32i udiv WR[wt]32i+31..32i
    endfor

DIV_U.D
    for i in 0 .. WRLEN/64-1
        WR[wd]64i+63..64i ← WR[ws]64i+63..64i udiv WR[wt]64i+63..64i
    endfor
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception.




**Format:** `DLSA`

`DLSA rd,rs,rt,sa` **MSA**

**Purpose:** Doubleword Left Shift Add

To left-shift a doubleword by a fixed number of bits and add the result to another doubleword.

**Description:** `GPR[rd] ← (GPR[rs] << (sa + 1)) + GPR[rt]`

The 64-bit doubleword value in GPR *rs* is shifted left, inserting zeros into the emptied bits; the 64-bit doubleword result is added to the 64-bit value in GPR *rt* and the 64-bit arithmetic result is placed into GPR *rd*.

No Integer Overflow exception occurs under any circumstances.

**Restrictions:**

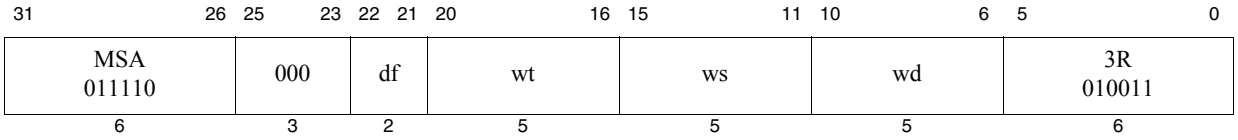
A Reserved Instruction Exception is signaled if access to 64-bit operations is not enabled or MSA implementation is not present.

**Operation:**

```
if Are64bitOperationsEnabled() and Config3MSAP = 1 then
   s ← sa + 1
   temp ← (GPR[rs](63-s)..0 || 0s) + GPR[rt]
   GPR[rd] ← temp63..0
else
   SignalException(ReservedInstruction)
endif
```

**Exceptions:**

Reserved Instruction Exception.




**Format:** `DOTP_S.df`

`DOTP_S.H wd,ws,wt` **MSA**

`DOTP_S.W wd,ws,wt` **MSA**

`DOTP_S.D wd,ws,wt` **MSA**

**Purpose:** Vector Signed Dot Product

Vector signed dot product (multiply and then pairwise add the adjacent multiplication results) to double width elements.

**Description:** `(wd[2i+1], wd[2i]) ← signed(ws[2i+1]) * signed(wt[2i+1]) + signed(ws[2i]) * signed(wt[2i])`

The signed integer elements in vector *wt* are multiplied by signed integer elements in vector *ws* producing a result twice the size of the input operands. The multiplication results of adjacent odd/even elements are added and stored to the destination.

The operands are values in integer data format half the size of *df*. The results are values in integer data format *df*.

**Restrictions:**

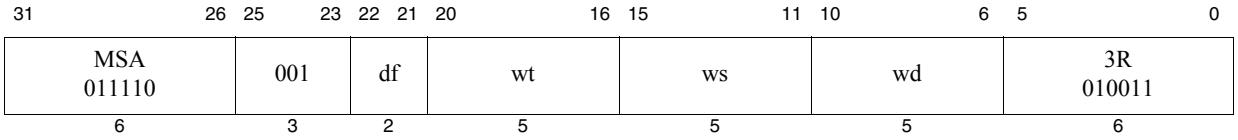
No data-dependent exceptions are possible.

**Operation:**

```
DOTP_S.H
    for i in 0 .. WRLEN/16-1
        WR[wd]16i+15..16i ← dotp_s(WR[ws]16i+15..16i, WR[wt]16i+15..16i, 8)
    endfor

DOTP_S.W
    for i in 0 .. WRLEN/32-1
        WR[wd]32i+31..32i ← dotp_s(WR[ws]32i+31..32i, WR[wt]32i+31..32i, 16)
    endfor

DOTP_S.D
    for i in 0 .. WRLEN/64-1
        WR[wd]64i+63..64i ← dotp_s(WR[ws]64i+63..64i, WR[wt]64i+63..64i, 32)
    endfor

function mulx_s(ts, tt, n)
    s ← (tsn-1)n || tsn-1..0
    t ← (ttn-1)n || ttn-1..0
    p ← s * t
    return p2n-1..0
endfunction mulx_s

function dotp_s(ts, tt, n)
    p1 ← mulx_s(ts2n-1..n, tt2n-1..n, n)
    p0 ← mulx_s(tsn-1..0, ttn-1..0, n)
    p ← p1 + p0
    return p2n-1..0
endfunction dotp_s
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception.




**Format:** `DOTP_U.df`

`DOTP_U.H wd,ws,wt` **MSA**

`DOTP_U.W wd,ws,wt` **MSA**

`DOTP_U.D wd,ws,wt` **MSA**

**Purpose:** Vector Unsigned Dot Product

Vector unsigned dot product (multiply and then pairwise add the adjacent multiplication results) to double width elements.

**Description:** `(wd[2i+1], wd[2i]) ← unsigned(ws[2i+1]) * unsigned(wt[2i+1]) + unsigned(ws[2i]) * unsigned(wt[2i])`

The unsigned integer elements in vector *wt* are multiplied by unsigned integer elements in vector *ws* producing a result twice the size of the input operands. The multiplication results of adjacent odd/even elements are added and stored to the destination.

The operands are values in integer data format half the size of *df*. The results are values in integer data format *df*.

**Restrictions:**

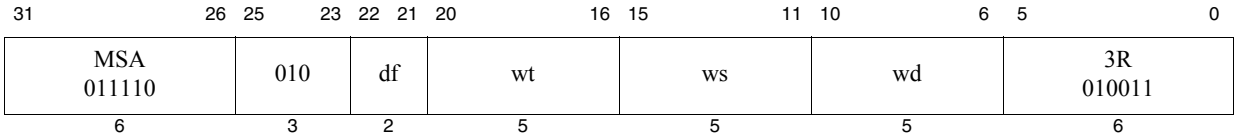
No data-dependent exceptions are possible.

**Operation:**

```
DOTP_U.H
    for i in 0 .. WRLEN/16-1
        WR[wd]16i+15..16i ← dotp_u(WR[ws]16i+15..16i, WR[wt]16i+15..16i, 8)
    endfor

DOTP_U.W
    for i in 0 .. WRLEN/32-1
        WR[wd]32i+31..32i ← dotp_u(WR[ws]32i+31..32i, WR[wt]32i+31..32i, 16)
    endfor

DOTP_U.D
    for i in 0 .. WRLEN/64-1
        WR[wd]64i+63..64i ← dotp_u(WR[ws]64i+63..64i, WR[wt]64i+63..64i, 32)
    endfor

function mulx_u(ts, tt, n)
    s ← 0^n || ts n-1..0
    t ← 0^n || tt n-1..0
    p ← s * t
    return p 2n-1..0
endfunction mulx_s

function dotp_u(ts, tt, n)
    p1 ← mulx_u(ts 2n-1..n, tt 2n-1..n, n)
    p0 ← mulx_u(ts n-1..0, tt n-1..0, n)
    p ← p1 + p0
    return p 2n-1..0
endfunction dotp_u
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception.




**Format:** `DPADD_S.df`

`DPADD_S.H wd,ws,wt` **MSA**

`DPADD_S.W wd,ws,wt` **MSA**

`DPADD_S.D wd,ws,wt` **MSA**

**Purpose:** Vector Signed Dot Product and Add

Vector signed dot product (multiply and then pairwise add the adjacent multiplication results) and add to double width elements.

**Description:** `(wd[2i+1], wd[2i]) ← (wd[2i+1], wd[2i]) + signed(ws[2i+1]) * signed(wt[2i+1]) + signed(ws[2i]) * signed(wt[2i])`

The signed integer elements in vector *wt* are multiplied by signed integer elements in vector *ws* producing a result twice the size of the input operands. The multiplication results of adjacent odd/even elements are added to the integer elements in vector *wd*.

The operands are values in integer data format half the size of *df*. The results are values in integer data format *df*.

**Restrictions:**

No data-dependent exceptions are possible.

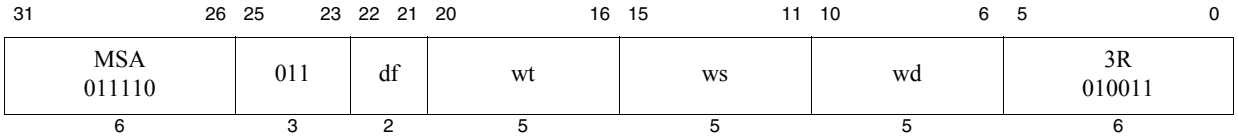
**Operation:**

```
DPADD_S.H
   for i in 0 .. WRLEN/16-1
      WR[wd]16i+15..16i ←
            WR[wd]16i+15..16i + dotp_s(WR[ws]16i+15..16i, WR[wt]16i+15..16i, 8)
   endfor

DPADD_S.W
   for i in 0 .. WRLEN/32-1
      WR[wd]32i+31..32i ←
            WR[wd]32i+31..32i + dotp_s(WR[ws]32i+31..32i, WR[wt]32i+31..32i, 16)
   endfor

DPADD_S.D
   for i in 0 .. WRLEN/64-1
      WR[wd]64i+63..64i ←
            WR[wd]64i+63..64i + dotp_s(WR[ws]64i+63..64i, WR[wt]64i+63..64i, 32)
   endfor

function mulx_s(ts, tt, n)
   s ← (tsn-1)n || tsn-1..0
   t ← (ttn-1)n || ttn-1..0
   p ← s * t
   return p2n-1..0
endfunction mulx_s

function dotp_s(ts, tt, n)
   p1 ← mulx_s(ts2n-1..n, tt2n-1..n, n)
   p0 ← mulx_s(tsn-1..0, ttn-1..0, n)
```

```
    p ← p1 + p0
    return p2n-1..0
endfunction dotp_s
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception.




**Format:** `DPADD_U.df`

`DPADD_U.H wd,ws,wt` **MSA**

`DPADD_U.W wd,ws,wt` **MSA**

`DPADD_U.D wd,ws,wt` **MSA**

**Purpose:** Vector Unsigned Dot Product and Add

Vector unsigned dot product (multiply and then pairwise add the adjacent multiplication results) and add to double width results.

**Description:** `(wd[2i+1], wd[2i]) ← (wd[2i+1], wd[2i]) + unsigned(ws[2i+1]) * unsigned(wt[2i+1]) + unsigned(ws[2i]) * unsigned(wt[2i])`

The unsigned integer elements in vector *wt* are multiplied by unsigned integer elements in vector *ws* producing a result twice the size of the input operands. The multiplication results of adjacent odd/even elements are added to the integer elements in vector *wd*.

The operands are values in integer data format half the size of *df*. The results are values in integer data format *df*.

**Restrictions:**

No data-dependent exceptions are possible.

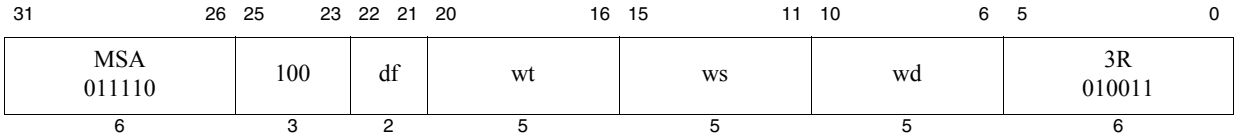
**Operation:**

```
DPADD_U.H
    for i in 0 .. WRLEN/16-1
        WR[wd]16i+15..16i ←
                WR[wd]16i+15..16i + dotp_u(WR[ws]16i+15..16i, WR[wt]16i+15..16i, 8)
    endfor

DPADD_U.W
    for i in 0 .. WRLEN/32-1
        WR[wd]32i+31..32i ←
                WR[wd]32i+31..32i + dotp_u(WR[ws]32i+31..32i, WR[wt]32i+31..32i, 16)
    endfor

DPADD_U.D
    for i in 0 .. WRLEN/64-1
        WR[wd]64i+63..64i ←
                WR[wd]64i+63..64i + dotp_u(WR[ws]64i+63..64i, WR[wt]64i+63..64i, 32)
    endfor

function mulx_u(ts, tt, n)
    s ← 0^n || tsn-1..0
    t ← 0^n || ttn-1..0
    p ← s * t
    return p2n-1..0
endfunction mulx_s

function dotp_u(ts, tt, n)
    p1 ← mulx_u(ts2n-1..n, tt2n-1..n, n)
    p0 ← mulx_u(tsn-1..0, ttn-1..0, n)
```

```
    p ← p1 + p0
    return p2n-1..0
endfunction dotp_u
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception.




**Format:** `DPSUB_S.df`

`DPSUB_S.H wd,ws,wt` **MSA**

`DPSUB_S.W wd,ws,wt` **MSA**

`DPSUB_S.D wd,ws,wt` **MSA**

**Purpose:** Vector Signed Dot Product and Subtract

Vector signed dot product (multiply and then pairwise add the adjacent multiplication results) and subtract from double width elements.

**Description:** `(wd[2i+1], wd[2i]) ← (wd[2i+1], wd[2i]) - (signed(ws[2i+1]) * signed(wt[2i+1]) + signed(ws[2i]) * signed(wt[2i]))`

The signed integer elements in vector *wt* are multiplied by signed integer elements in vector *ws* producing a signed result twice the size of the input ope rands. The sum of multiplication results of adjacent odd/even elements is subtracted from the integer elements in vector *wd* to a signed result.

The operands are values in integer data format half the size of *df*. The results are values in integer data format *df*.

**Restrictions:**

No data-dependent exceptions are possible.

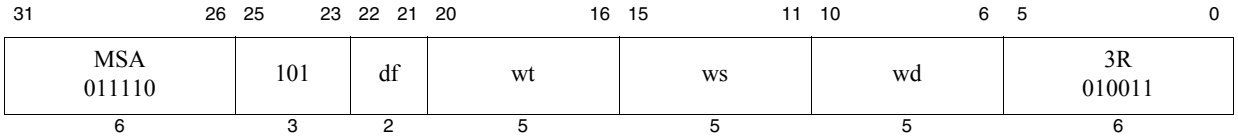
**Operation:**

```
DPSUB_S.H
   for i in 0 .. WRLEN/16-1
      WR[wd]16i+15..16i ←
            WR[wd]16i+15..16i - dotp_s(WR[ws]16i+15..16i, WR[wt]16i+15..16i, 8)
   endfor

DPSUB_S.W
   for i in 0 .. WRLEN/32-1
      WR[wd]32i+31..32i ←
            WR[wd]32i+31..32i - dotp_s(WR[ws]32i+31..32i, WR[wt]32i+31..32i, 16)
   endfor

DPSUB_S.D
   for i in 0 .. WRLEN/64-1
      WR[wd]64i+63..64i ←
            WR[wd]64i+63..64i - dotp_s(WR[ws]64i+63..64i, WR[wt]64i+63..64i, 32)
   endfor

function mulx_s(ts, tt, n)
   s ← (tsn-1)n || tsn-1..0
   t ← (ttn-1)n || ttn-1..0
   p ← s * t
   return p2n-1..0
endfunction mulx_s

function dotp_s(ts, tt, n)
   p1 ← mulx_s(ts2n-1..n, tt2n-1..n, n)
   p0 ← mulx_s(tsn-1..0, ttn-1..0, n)
```

```
    p ← p1 + p0
    return p2n-1..0
endfunction dotp_s
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception.




**Format:** `DPSUB_U.df`

`DPSUB_U.H wd,ws,wt` **MSA**

`DPSUB_U.W wd,ws,wt` **MSA**

`DPSUB_U.D wd,ws,wt` **MSA**

**Purpose:** Vector Unsigned Dot Product and Subtract

Vector unsigned dot product (multiply and then pairwise add the adjacent multiplication results) and subtract from double width elements.

**Description:** `(wd[2i+1], wd[2i]) ← (wd[2i+1], wd[2i]) - (unsigned(ws[2i+1]) * unsigned(wt[2i+1]) + unsigned(ws[2i]) * unsigned(wt[2i]))`

The unsigned integer elements in vector *wt* are multiplied by unsigned integer elements in vector *ws* producing a positive, unsigned result twice the size of the input operands. The sum of multiplication results of adjacent odd/even elements is subtracted from the integer elements in vector *wd* to a signed result.

The operands are values in integer data format half the size of *df*. The results are values in integer data format *df*.

**Restrictions:**

No data-dependent exceptions are possible.

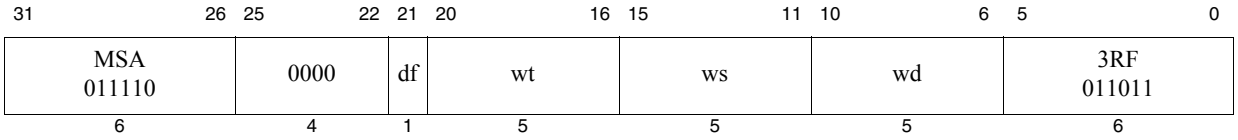
**Operation:**

```
DPSUB_U.H
    for i in 0 .. WRLEN/16-1
        WR[wd]16i+15..16i ←
                WR[wd]16i+15..16i - dotp_u(WR[ws]16i+15..16i, WR[wt]16i+15..16i, 8)
    endfor

DPSUB_U.W
    for i in 0 .. WRLEN/32-1
        WR[wd]32i+31..32i ←
                WR[wd]32i+31..32i - dotp_u(WR[ws]32i+31..32i, WR[wt]32i+31..32i, 16)
    endfor

DPSUB_U.D
    for i in 0 .. WRLEN/64-1
        WR[wd]64i+63..64i ←
                WR[wd]64i+63..64i - dotp_u(WR[ws]64i+63..64i, WR[wt]64i+63..64i, 32)
    endfor

function mulx_u(ts, tt, n)
    s ← 0^n || tsn-1..0
    t ← 0^n || ttn-1..0
    p ← s * t
    return p2n-1..0
endfunction mulx_s

function dotp_u(ts, tt, n)
    p1 ← mulx_u(ts2n-1..n, tt2n-1..n, n)
    p0 ← mulx_u(tsn-1..0, ttn-1..0, n)
```

```
    p ← p1 + p0
    return p2n-1..0
endfunction dotp_u
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception.




**Format:** `FADD.df`

`FADD.W wd,ws,wt` **MSA**

`FADD.D wd,ws,wt` **MSA**

**Purpose:** Vector Floating-Point Addition

Vector floating-point addition.

**Description:** `wd[i] ← ws[i] + wt[i]`

The floating-point elements in vector *wt* are added to the floating-point elements in vector *ws*. The result is written to vector *wd*.

The add operation is defined by the IEEE Standard for Floating-Point Arithmetic 754™-2008.

The operands and results are values in floating-point data format *df*.

**Restrictions:**

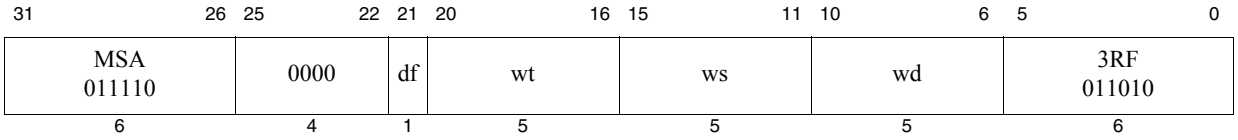
Data-dependent exceptions are possible as specified by the IEEE Standard for Floating-Point Arithmetic 754™-2008.

**Operation:**

```
FADD.W
   for i in 0 .. WRLEN/32-1
      WR[wd]32i+31..32i ← AddFP(WR[ws]32i+31..32i, WR[wt]32i+31..32i, 32)
   endfor

FADD.D
   for i in 0 .. WRLEN/64-1
      WR[wd]64i+63..64i ← AddFP(WR[ws]64i+63..64i, WR[wt]64i+63..64i, 64)
   endfor

function AddFP(tt, ts, n)
   /* Implementation defined add operation. */
endfunction AddFP
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception, MSA Floating Point Exception.




**Format:** `FCAF.df`

`FCAF.W wd,ws,wt` **MSA**

`FCAF.D wd,ws,wt` **MSA**

**Purpose:** Vector Floating-Point Quiet Compare Always False

Vector to vector floating-point quiet compare always false; all destination bits are clear.

**Description:** `wd[i] ← quietFalse(ws[i], wt[i])`

Set all bits to 0 in *wd* elements. Signaling NaN elements in *ws* or *wt* signal Invalid Operation exception.

The Inexact Exception is not signaled when subnormal input operands are flushed based on the flush-to-zero bit FS in MSA Control and Status Register *MSACSR*. In case of a floating-point exception, the default result has all bits set to 0.

The operands are values in floating-point data format *df*. The results are values in integer data format *df*.

**Restrictions:**

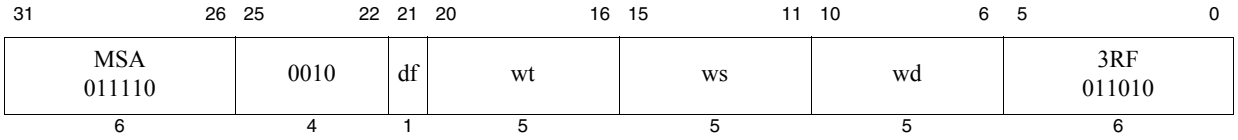
Data-dependent exceptions are possible as specified by the IEEE Standard for Floating-Point Arithmetic 754™-2008.

**Operation:**

```
FCAF.W
   for i in 0 .. WRLEN/32-1
      WR[wd]32i+31..32i ← QuietFALSE(WR[ws]32i+31..32i, WR[wt]32i+31..32i, 32)
   endfor

FCAF.D
   for i in 0 .. WRLEN/64-1
      WR[wd]64i+63..64i ← QuietFALSE(WR[ws]64i+63..64i, WR[wt]64i+63..64i, 64)
   endfor

function QuietFALSE(tt, ts, n)
   /* Implementation defined signaling NaN test */
   return 0
endfunction QuietFALSE
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception, MSA Floating Point Exception.




**Format:** `FCEQ.df`

`FCEQ.W wd,ws,wt` **MSA**

`FCEQ.D wd,ws,wt` **MSA**

**Purpose:** Vector Floating-Point Quiet Compare Equal

Vector to vector floating-point quiet compare for equality; if true all destination bits are set, otherwise clear.

**Description:** `wd[i] ← (ws[i] =(quiet) wt[i])`

Set all bits to 1 in *wd* elements if the corresponding *ws* and *wt* floating-point elements are ordered and equal, otherwise set all bits to 0.

The quiet compare operation is defined by the IEEE Standard for Floating-Point Arithmetic 754™-2008.

The Inexact Exception is not signaled when subnormal input operands are flushed based on the flush-to-zero bit FS in MSA Control and Status Register *MSACSR*. In case of a floating-point exception, the default result has all bits set to 0.

The operands are values in floating-point data format *df*. The results are values in integer data format *df*.

**Restrictions:**

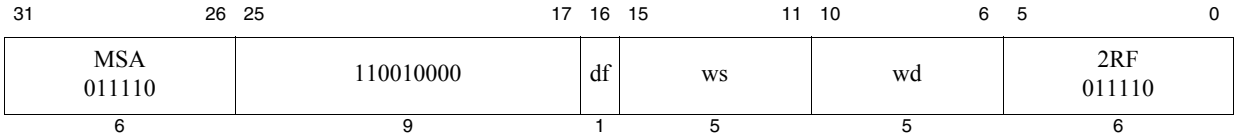
Data-dependent exceptions are possible as specified by the IEEE Standard for Floating-Point Arithmetic 754™-2008.

**Operation:**

```
FCEQ.W
    for i in 0 .. WRLEN/32-1
        c ← EqualFP(WR[ws]32i+31..32i, WR[wt]32i+31..32i, 32)
        WR[wd]32i+31..32i ← c32
    endfor

FCEQ.D
    for i in 0 .. WRLEN/64-1
        c ← EqualFP(WR[ws]64i+63..64i, WR[wt]64i+63..64i, 64)
        WR[wd]64i+63..64i ← c64
    endfor

function EqualFP(tt, ts, n)
    /* Implementation defined quiet equal compare operation. */
endfunction EqualFP
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception, MSA Floating Point Exception.




**Format:** `FCLASS.df`

`FCLASS.W wd,ws` **MSA**

`FCLASS.D wd,ws` **MSA**

**Purpose:** Vector Floating-Point Class Mask

Vector floating-point class shown as a bit mask for Zero, Negative, Infinite, Subnormal, Quiet NaN, or Signaling NaN.

**Description:** `wd[i] ← class(ws[i])`

Store in each element of vector *wd* a bit mask reflecting the floating-point class of the corresponding element of vector *ws*.

The mask has 10 bits as follows. Bits 0 and 1 indicate NaN values: signaling NaN (bit 0) and quiet NaN (bit 1). Bits 2, 3, 4, 5 classify negative values: infinity (bit 2), normal (bit 3), subnormal (bit 4), and zero (bit 5). Bits 6, 7, 8, 9 classify positive values:infinity (bit 6), normal (bit 7), subnormal (bit 8), and zero (bit 9).

The input values and generated bit masks are not affected by the flush-to-zero bit FS in MSA Control and Status Register *MSACSR*.

The operands are values in floating-point data format *df*. The results are values in integer data format *df*.

**Restrictions:**

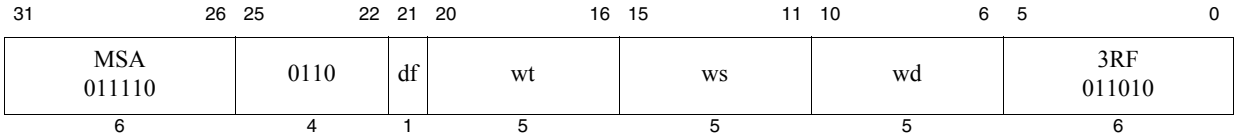
No data-dependent exceptions are possible.

**Operation:**

```
FCLASS.W
   for i in 0 .. WRLEN/32-1
      c ← ClassFP(WR[ws]32i+31..32i, 32)
      WR[wd]32i+31..32i ← 0^22 || c9..0
   endfor

FCLASS.D
   for i in 0 .. WRLEN/64-1
      c ← ClassFP(WR[ws]64i+63..64i, 64)
      WR[wd]64i+63..64i ← 0^54 || c9..0
   endfor

function ClassFP(tt, n)
   /* Implementation defined class operation. */
endfunction ClassFP
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception.




**Format:** `FCLE.df`

`FCLE.W wd,ws,wt` **MSA**

`FCLE.D wd,ws,wt` **MSA**

**Purpose:** Vector Floating-Point Quiet Compare Less or Equal

Vector to vector floating-point quiet compare for less than or equal; if true all destination bits are set, otherwise clear.

**Description:** `wd[i] ← (ws[i] <=(quiet) wt[i])`

Set all bits to 1 in *wd* elements if the corresponding *ws* floating-point elements are ordered and either less than or equal to *wt* floating-point elements, otherwise set all bits to 0.

The quiet compare operation is defined by the IEEE Standard for Floating-Point Arithmetic 754™-2008.

The Inexact Exception is not signaled when subnormal input operands are flushed based on the flush-to-zero bit FS in MSA Control and Status Register *MSACSR*. In case of a floating-point exception, the default result has all bits set to 0.

The operands are values in floating-point data format *df*. The results are values in integer data format *df*.

**Restrictions:**

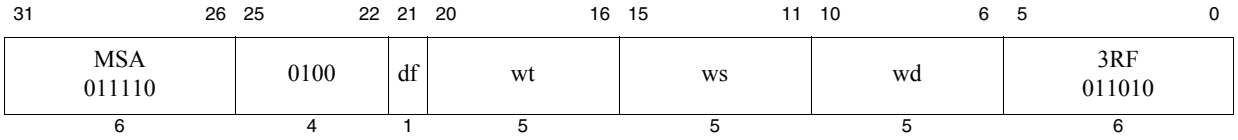
Data-dependent exceptions are possible as specified by the IEEE Standard for Floating-Point Arithmetic 754™-2008.

**Operation:**

```
FCLE.W
   for i in 0 .. WRLEN/32-1
      c ← LessFP(WR[ws]32i+31..32i, WR[wt]32i+31..32i, 32)
      d ← EqualFP(WR[ws]32i+31..32i, WR[wt]32i+31..32i, 32)
      WR[wd]32i+31..32i ← (c | d)32
   endfor

FCLE.D
   for i in 0 .. WRLEN/64-1
      c ← LessFP(WR[ws]64i+63..64i, WR[wt]64i+63..64i, 64)
      d ← EqualFP(WR[ws]64i+63..64i, WR[wt]64i+63..64i, 64)
      WR[wd]64i+63..64i ← (c | d)64
   endfor

function LessThanFP(tt, ts, n)
   /* Implementation defined quiet less than compare operation. */
endfunction LessThanFP

function EqualFP(tt, ts, n)
   /* Implementation defined quiet equal compare operation. */
endfunction EqualFP
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception, MSA Floating Point Exception.




**Format:** `FCLT.df`

`FCLT.W wd,ws,wt` **MSA**

`FCLT.D wd,ws,wt` **MSA**

**Purpose:** Vector Floating-Point Quiet Compare Less Than

Vector to vector floating-point quiet compare for less than; if true all destination bits are set, otherwise clear.

**Description:** `wd[i] ← (ws[i] <(quiet) wt[i])`

Set all bits to 1 in *wd* elements if the corresponding *ws* floating-point elements are ordered and less than *wt* floating-point elements, otherwise set all bits to 0.

The quiet compare operation is defined by the IEEE Standard for Floating-Point Arithmetic 754™-2008.

The Inexact Exception is not signaled when subnormal input operands are flushed based on the flush-to-zero bit FS in MSA Control and Status Register *MSACSR*. In case of a floating-point exception, the default result has all bits set to 0.

The operands are values in floating-point data format *df*. The results are values in integer data format *df*.

**Restrictions:**

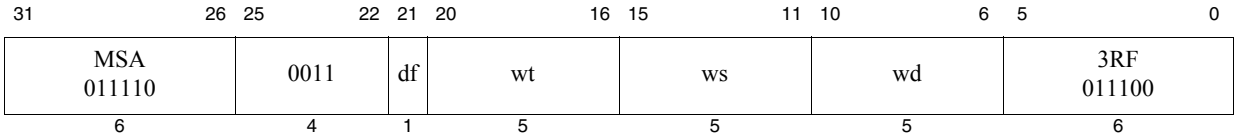
Data-dependent exceptions are possible as specified by the IEEE Standard for Floating-Point Arithmetic 754™-2008.

**Operation:**

```
FCLT.W
    for i in 0 .. WRLEN/32-1
        c ← LessFP(WR[ws]32i+31..32i, WR[wt]32i+31..32i, 32)
        WR[wd]32i+31..32i ← c32
    endfor

FCLT.D
    for i in 0 .. WRLEN/64-1
        c ← LessFP(WR[ws]64i+63..64i, WR[wt]64i+63..64i, 64)
        WR[wd]64i+63..64i ← c64
    endfor

function LessThanFP(tt, ts, n)
    /* Implementation defined quiet less than compare operation. */
endfunction LessThanFP
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception, MSA Floating Point Exception.




**Format:** `FCNE.df`

`FCNE.W wd,ws,wt` **MSA**

`FCNE.D wd,ws,wt` **MSA**

**Purpose:** Vector Floating-Point Quiet Compare Not Equal

Vector to vector floating-point quiet compare for not equal; if true all destination bits are set, otherwise clear.

**Description:** `wd[i] ← (ws[i] ≠(quiet) wt[i])`

Set all bits to 1 in *wd* elements if the corresponding *ws* and *wt* floating-point elements are ordered and not equal, otherwise set all bits to 0.

The quiet compare operation is defined by the IEEE Standard for Floating-Point Arithmetic 754™-2008.

The Inexact Exception is not signaled when subnormal input operands are flushed based on the flush-to-zero bit FS in MSA Control and Status Register *MSACSR*. In case of a floating-point exception, the default result has all bits set to 0.

The operands are values in floating-point data format *df*. The results are values in integer data format *df*.

**Restrictions:**

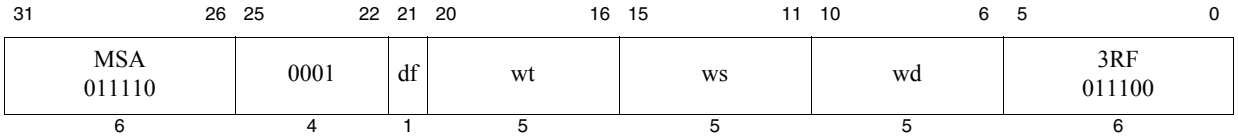
Data-dependent exceptions are possible as specified by the IEEE Standard for Floating-Point Arithmetic 754™-2008.

**Operation:**

```
FCNE.W
    for i in 0 .. WRLEN/32-1
        c ← NotEqualFP(WR[ws]32i+31..32i, WR[wt]32i+31..32i, 32)
        WR[wd]32i+31..32i ← c32
    endfor

FCNE.D
    for i in 0 .. WRLEN/64-1
        c ← NotEqualFP(WR[ws]64i+63..64i, WR[wt]64i+63..64i, 64)
        WR[wd]64i+63..64i ← c64
    endfor

function NotEqualFP(tt, ts, n)
    /* Implementation defined quiet not equal compare operation. */
endfunction NotEqualFP
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception, MSA Floating Point Exception.




**Format:** `FCOR.df`

`FCOR.W wd,ws,wt` **MSA**

`FCOR.D wd,ws,wt` **MSA**

**Purpose:** Vector Floating-Point Quiet Compare Ordered

Vector to vector floating-point quiet compare ordered; if true all destination bits are set, otherwise clear.

**Description:** `wd[i] ← ws[i] !?(quiet) wt[i]`

Set all bits to 1 in *wd* elements if the corresponding *ws* and *wt* floating-point elements are ordered, i.e. both elements are not NaN values, otherwise set all bits to 0.

The quiet compare operation is defined by the IEEE Standard for Floating-Point Arithmetic 754$^{TM}$-2008.

The Inexact Exception is not signaled when subnormal input operands are flushed based on the flush-to-zero bit FS in MSA Control and Status Register *MSACSR*. In case of a floating-point exception, the default result has all bits set to 0.

The operands are values in floating-point data format *df*. The results are values in integer data format *df*.

**Restrictions:**

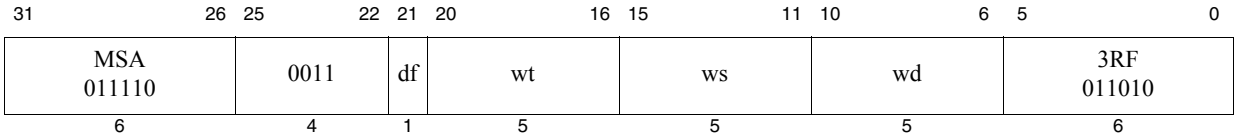
Data-dependent exceptions are possible as specified by the IEEE Standard for Floating-Point Arithmetic 754$^{TM}$-2008.

**Operation:**

```
FCOR.W
    for i in 0 .. WRLEN/32-1
        c ←OrderedFP(WR[ws]32i+31..32i, WR[wt]32i+31..32i, 32)
        WR[wd]32i+31..32i ← c32
    endfor

FCOR.D
    for i in 0 .. WRLEN/64-1
        c ← OrderedFP(WR[ws]64i+63..64i, WR[wt]64i+63..64i, 64)
        WR[wd]64i+63..64i ← c64
    endfor

function OrderedFP(tt, ts, n)
    /* Implementation defined quiet ordered compare operation. */
endfunction OrderedFP
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception, MSA Floating Point Exception.




**Format:** `FCUEQ.df`

`FCUEQ.W wd,ws,wt` **MSA**

`FCUEQ.D wd,ws,wt` **MSA**

**Purpose:** Vector Floating-Point Quiet Compare Unordered or Equal

Vector to vector floating-point quiet compare for unordered or equality; if true all destination bits are set, otherwise clear.

**Description:** `wd[i] ← (ws[i] =?(quiet) wt[i])`

Set all bits to 1 in *wd* elements if the corresponding *ws* and *wt* floating-point elements are unordered or equal, otherwise set all bits to 0.

The quiet compare operation is defined by the IEEE Standard for Floating-Point Arithmetic 754™-2008.

The Inexact Exception is not signaled when subnormal input operands are flushed based on the flush-to-zero bit FS in MSA Control and Status Register *MSACSR*. In case of a floating-point exception, the default result has all bits set to 1.

The operands are values in floating-point data format *df*. The results are values in integer data format *df*.

**Restrictions:**

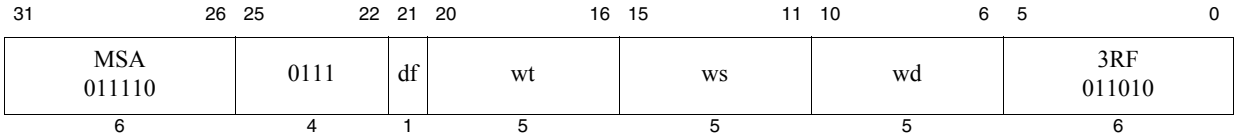
Data-dependent exceptions are possible as specified by the IEEE Standard for Floating-Point Arithmetic 754™-2008.

**Operation:**

```
FCUEQ.W
    for i in 0 .. WRLEN/32-1
        c ← UnorderedFP(WR[ws]32i+31..32i, WR[wt]32i+31..32i, 32)
        d ← EqualFP(WR[ws]32i+31..32i, WR[wt]32i+31..32i, 32)
        WR[wd]32i+31..32i ← (c | d)32
    endfor

FCUEQ.D
    for i in 0 .. WRLEN/64-1
        c ← UnorderedFP(WR[ws]64i+63..64i, WR[wt]64i+63..64i, 64)
        d ← EqualFP(WR[ws]64i+63..64i, WR[wt]64i+63..64i, 64)
        WR[wd]64i+63..64i ← (c | d)64
    endfor

function UnorderedFP(tt, ts, n)
    /* Implementation defined quiet unordered compare operation. */
endfunction UnorderedFP

function EqualFP(tt, ts, n)
    /* Implementation defined quiet equal compare operation. */
endfunction EqualFP
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception, MSA Floating Point Exception.




**Format:** `FCULE.df`

`FCULE.W wd,ws,wt` **MSA**

`FCULE.D wd,ws,wt` **MSA**

**Purpose:** Vector Floating-Point Quiet Compare Unordered or Less or Equal

Vector to vector floating-point quiet compare for unordered or less than or equal; if true all destination bits are set, otherwise clear.

**Description:** `wd[i] ← (ws[i] <=?(quiet) wt[i])`

Set all bits to 1 in *wd* elements if the corresponding *ws* floating-point elements are unordered or less than or equal to *wt* floating-point elements, otherwise set all bits to 0.

The quiet compare operation is defined by the IEEE Standard for Floating-Point Arithmetic 754™-2008.

The Inexact Exception is not signaled when subnormal input operands are flushed based on the flush-to-zero bit FS in MSA Control and Status Register *MSACSR*. In case of a floating-point exception, the default result has all bits set to 1.

The operands are values in floating-point data format *df*. The results are values in integer data format *df*.

**Restrictions:**

Data-dependent exceptions are possible as specified by the IEEE Standard for Floating-Point Arithmetic 754™-2008.

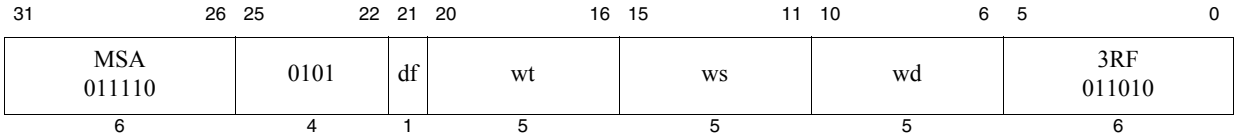
**Operation:**

```
FCULE.W
    for i in 0 .. WRLEN/32-1
        c ← UnorderedFP(WR[ws]32i+31..32i, WR[wt]32i+31..32i, 32)
        d ← LessFP(WR[ws]32i+31..32i, WR[wt]32i+31..32i, 32)
        e ← EqualFP(WR[ws]32i+31..32i, WR[wt]32i+31..32i, 32)
        WR[wd]32i+31..32i ← (c | d | e)32
    endfor

FCULE.D
    for i in 0 .. WRLEN/64-1
        c ← UnorderedFP(WR[ws]64i+63..64i, WR[wt]64i+63..64i, 64)
        d ← LessFP(WR[ws]64i+63..64i, WR[wt]64i+63..64i, 64)
        e ← EqualFP(WR[ws]64i+63..64i, WR[wt]64i+63..64i, 64)
        WR[wd]64i+63..64i ← (c | d | e)64
    endfor

function UnorderedFP(tt, ts, n)
    /* Implementation defined quiet unordered compare operation. */
endfunction UnorderedFP

function LessThanFP(tt, ts, n)
    /* Implementation defined quiet less than compare operation. */
endfunction LessThanFP
```

```
function EqualFP(tt, ts, n)
   /* Implementation defined quiet equal compare operation. */
endfunction EqualFP
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception, MSA Floating Point Exception.




**Format:** `FCULT.df`

`FCULT.W wd,ws,wt` **MSA**

`FCULT.D wd,ws,wt` **MSA**

**Purpose:** Vector Floating-Point Quiet Compare Unordered or Less Than

Vector to vector floating-point quiet compare for unordered or less than; if true all destination bits are set, otherwise clear.

**Description:** `wd[i] ← (ws[i] <?(quiet) wt[i])`

Set all bits to 1 in *wd* elements if the corresponding *ws* floating-point elements are unordered or less than *wt* floating-point elements, otherwise set all bits to 0.

The quiet compare operation is defined by the IEEE Standard for Floating-Point Arithmetic 754™-2008.

The Inexact Exception is not signaled when subnormal input operands are flushed based on the flush-to-zero bit FS in MSA Control and Status Register *MSACSR*. In case of a floating-point exception, the default result has all bits set to 1.

The operands are values in floating-point data format *df*. The results are values in integer data format *df*.

**Restrictions:**

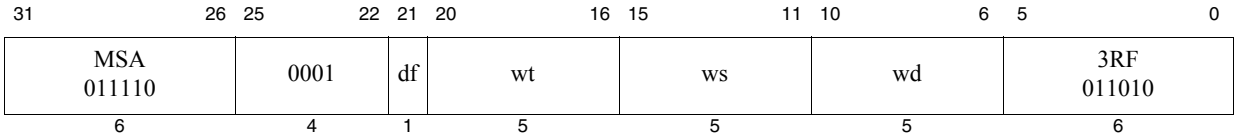
Data-dependent exceptions are possible as specified by the IEEE Standard for Floating-Point Arithmetic 754™-2008.

**Operation:**

```
FCULT.W
    for i in 0 .. WRLEN/32-1
        c ← UnorderedFP(WR[ws]32i+31..32i, WR[wt]32i+31..32i, 32)
        d ← LessFP(WR[ws]32i+31..32i, WR[wt]32i+31..32i, 32)
        WR[wd]32i+31..32i ← (c | d)32
    endfor

FCULT.D
    for i in 0 .. WRLEN/64-1
        c ← LessFP(WR[ws]64i+63..64i, WR[wt]64i+63..64i, 64)
        d ← UnorderedFP(WR[ws]64i+63..64i, WR[wt]64i+63..64i, 64)
        WR[wd]64i+63..64i ← c64
    endfor

function UnorderedFP(tt, ts, n)
    /* Implementation defined quiet unordered compare operation. */
endfunction UnorderedFP

function LessThanFP(tt, ts, n)
    /* Implementation defined quiet less than compare operation. */
endfunction LessThanFP
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception, MSA Floating Point Exception.




**Format:** `FCUN.df`

`FCUN.W wd,ws,wt` **MSA**

`FCUN.D wd,ws,wt` **MSA**

**Purpose:** Vector Floating-Point Quiet Compare Unordered

Vector to vector floating-point quiet compare unordered; if true all destination bits are set, otherwise clear.

**Description:** `wd[i] ← (ws[i] ?(quiet) wt[i])`

Set all bits to 1 in *wd* elements if the corresponding *ws* and *wt* floating-point elements are unordered, i.e. at least one element is a NaN value, otherwise set all bits to 0.

The quiet compare operation is defined by the IEEE Standard for Floating-Point Arithmetic 754™-2008.

The Inexact Exception is not signaled when subnormal input operands are flushed based on the flush-to-zero bit FS in MSA Control and Status Register *MSACSR*. In case of a floating-point exception, the default result has all bits set to 1.

The operands are values in floating-point data format *df*. The results are values in integer data format *df*.

**Restrictions:**

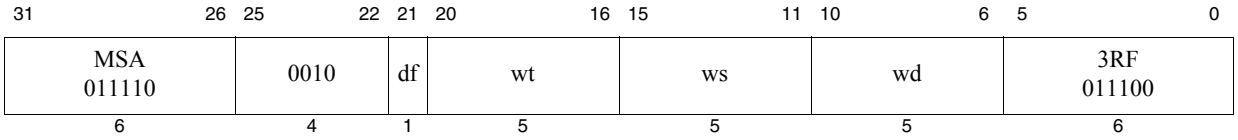
Data-dependent exceptions are possible as specified by the IEEE Standard for Floating-Point Arithmetic 754™-2008.

**Operation:**

```
FCUN.W
    for i in 0 .. WRLEN/32-1
        c ← UnorderedFP(WR[ws]32i+31..32i, WR[wt]32i+31..32i, 32)
        WR[wd]32i+31..32i ← c32
    endfor

FCUN.D
    for i in 0 .. WRLEN/64-1
        c ← UnorderedFP(WR[ws]64i+63..64i, WR[wt]64i+63..64i, 64)
        WR[wd]64i+63..64i ← c64
    endfor

function UnorderedFP(tt, ts, n)
    /* Implementation defined quiet unordered compare operation. */
endfunction UnorderedFP
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception, MSA Floating Point Exception.




**Format:** `FCUNE.df`

`FCUNE.W wd,ws,wt` **MSA**

`FCUNE.D wd,ws,wt` **MSA**

**Purpose:** Vector Floating-Point Quiet Compare Unordered or Not Equal

Vector to vector floating-point quiet compare for unordered or not equal; if true all destination bits are set, otherwise clear.

**Description:** `wd[i] ← (ws[i] ≠?(quiet) wt[i])`

Set all bits to 1 in *wd* elements if the corresponding *ws* and *wt* floating-point elements are unordered or not equal, otherwise set all bits to 0.

The quiet compare operation is defined by the IEEE Standard for Floating-Point Arithmetic 754™-2008.

The Inexact Exception is not signaled when subnormal input operands are flushed based on the flush-to-zero bit FS in MSA Control and Status Register *MSACSR*. In case of a floating-point exception, the default result has all bits set to 1.

The operands are values in floating-point data format *df*. The results are values in integer data format *df*.

**Restrictions:**

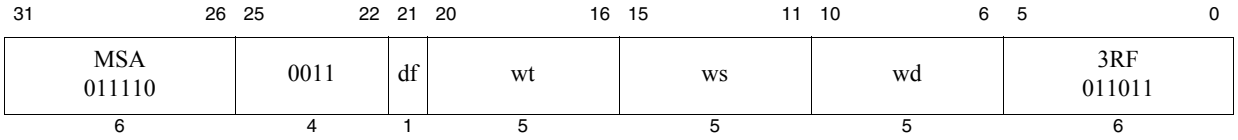
Data-dependent exceptions are possible as specified by the IEEE Standard for Floating-Point Arithmetic 754™-2008.

**Operation:**

```
FCUNE.W
    for i in 0 .. WRLEN/32-1
        c ← UnorderedFP(WR[ws]32i+31..32i, WR[wt]32i+31..32i, 32)
        d ← NotEqualFP(WR[ws]32i+31..32i, WR[wt]32i+31..32i, 32)
        WR[wd]32i+31..32i ← (c | d)32
    endfor

FCUNE.D
    for i in 0 .. WRLEN/64-1
        c ← UnorderedFP(WR[ws]64i+63..64i, WR[wt]64i+63..64i, 64)
        d ← NotEqualFP(WR[ws]64i+63..64i, WR[wt]64i+63..64i, 64)
        WR[wd]64i+63..64i ← (c | d)64
    endfor

function UnorderedFP(tt, ts, n)
    /* Implementation defined quiet unordered compare operation. */
endfunction UnorderedFP

function NotEqualFP(tt, ts, n)
    /* Implementation defined quiet not equal compare operation. */
endfunction NotEqualFP
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception, MSA Floating Point Exception.




**Format:** `FDIV.df`

`FDIV.W wd,ws,wt` **MSA**

`FDIV.D wd,ws,wt` **MSA**

**Purpose:** Vector Floating-Point Division

Vector floating-point division.

**Description:** `wd[i] ← ws[i] / wt[i]`

The floating-point elements in vector *ws* are divided by the floating-point elements in vector *wt*. The result is written to vector *wd*.

The divide operation is defined by the IEEE Standard for Floating-Point Arithmetic 754™-2008.

The operands and results are values in floating-point data format *df*.

**Restrictions:**

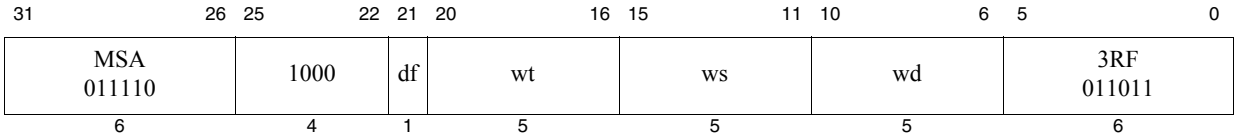
Data-dependent exceptions are possible as specified by the IEEE Standard for Floating-Point Arithmetic 754™-2008.

**Operation:**

```
FDIV.W
   for i in 0 .. WRLEN/32-1
      WR[wd]32i+31..32i ← DivideFP(WR[ws]32i+31..32i, WR[wt]32i+31..32i, 32)
   endfor

FDIV.D
   for i in 0 .. WRLEN/64-1
      WR[wd]64i+63..64i ← DivideFP(WR[ws]64i+63..64i, WR[wt]64i+63..64i, 64)
   endfor

function DivideFP(tt, ts, n)
   /* Implementation defined divide operation. */
endfunction DivideFP
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception, MSA Floating Point Exception.




**Format:** `FEXDO.df`

`FEXDO.H wd,ws,wt` **MSA**

`FEXDO.W wd,ws,wt` **MSA**

**Purpose:** Vector Floating-Point Down-Convert Interchange Format

Vector conversion to smaller interchange format.

**Description:** `left_half(wd)[i] ← down_convert(ws[i]); right_half(wd)[i] ← down_convert(wt[i])`

The floating-point elements in vectors *ws* and *wt* are down-converted to a smaller interchange format, i.e. from 64-bit to 32-bit, or from 32-bit to 16-bit.

The format down-conversion operation is defined by the IEEE Standard for Floating-Point Arithmetic 754™-2008.

16-bit floating-point results are not affected by the flush-to-zero bit FS in MSA Control and Status Register *MSACSR*.

The operands are values in floating-point data format double the size of *df*. The results are floating-point values in data format of *df*.

**Restrictions:**

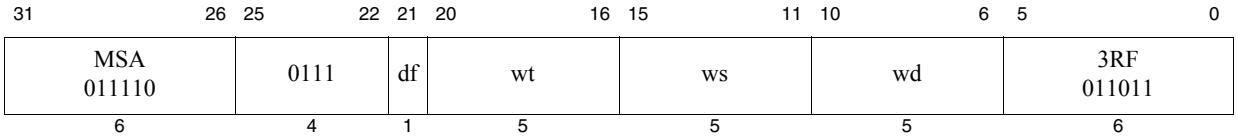
Data-dependent exceptions are possible as specified by the IEEE Standard for Floating-Point Arithmetic 754™-2008.

**Operation:**

```
FEXDO.H
    for i in 0 .. WRLEN/32-1
        f ← DownConvertFP(WR[ws]32i+31..32i, 32)
        g ← DownConvertFP(WR[wt]32i+31..32i, 32)
        WR[wd]16i+15+WRLEN/2..16i+WRLEN/2 ← f
        WR[wd]16i+15..16i ← g
    endfor

FEXDO.W
    for i in 0 .. WRLEN/64-1
        f ← DownConvertFP(WR[ws]64i+63..64i, 64)
        g ← DownConvertFP(WR[wt]64i+63..64i, 64)
        WR[wd]32i+31+WRLEN/2..32i+WRLEN/2 ← f
        WR[wd]32i+31..32i ← g
    endfor

function DownConvertFP(tt, n)
    /* Implementation defined format down-conversion. */
endfunction DownConvertFP
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception, MSA Floating Point Exception.




**Format:** `FEXP2.df`

`FEXP2.W wd,ws,wt` **MSA**

`FEXP2.D wd,ws,wt` **MSA**

**Purpose:** Vector Floating-Point Base 2 Exponentiation

Vector floating-point base 2 exponentiation.

**Description:** `wd[i] ← ws[i] * 2`$^{\mathbf{wt[i]}}$

The floating-point elements in vector *ws* are scaled, i.e. multiplied, by 2 to the power of integer elements in vector *wt*. The result is written to vector *wd*.

The operation is the homogeneous **scaleB()** as defined by the IEEE Standard for Floating-Point Arithmetic 754™-2008.

The *ws* operands and *wd* results are values in floating-point data format *df*. The *wt* operands are values in integer data format *df*.

**Restrictions:**

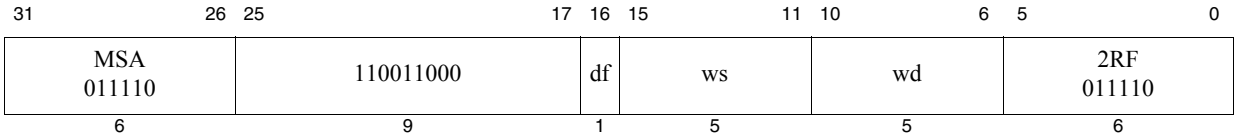
Data-dependent exceptions are possible as specified by the IEEE Standard for Floating-Point Arithmetic 754™-2008.

**Operation:**

```
FEXP2.W
   for i in 0 .. WRLEN/32-1
      WR[wd]32i+31..32i ← Exp2FP(WR[ws]32i+31..32i, WR[wt]32i+31..32i)
   endfor

FEXP2.D
   for i in 0 .. WRLEN/64-1
      WR[wd]64i+63..64i ← Exp2FP(WR[ws]64i+63..64i, WR[wt]64i+63..64i)
   endfor

function Exp2FP(tt, ts, n)
   /* Implementation defined tt * 2^ts operation. */
endfunction Exp2FP
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception, MSA Floating Point Exception.




**Format:** `FEXUPL.df`

`FEXUPL.W wd,ws` **MSA**

`FEXUPL.D wd,ws` **MSA**

**Purpose:** Vector Floating-Point Up-Convert Interchange Format Left

Vector left elements conversion to wider interchange format.

**Description:** `wd[i] ← up_convert(left_half(ws)[i])`

The left half floating-point elements in vector *ws* are up-converted to a larger interchange format, i.e. from 16-bit to 32-bit, or from 32-bit to 64-bit. The result is written to vector *wd*.

The format up-conversion operation is defined by the IEEE Standard for Floating-Point Arithmetic 754™-2008.

16-bit floating-point inputs are not affected by the flush-to-zero bit FS in MSA Control and Status Register *MSACSR*.

The operands are values in floating-point data format half the size of *df*. The results are floating-point values in data format *df*.

**Restrictions:**

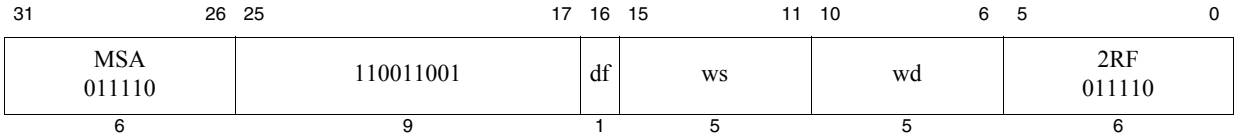
Data-dependent exceptions are possible as specified by the IEEE Standard for Floating-Point Arithmetic 754™-2008.

**Operation:**

```
FEXUPL.W
   for i in 0 .. WRLEN/32-1
      f ← UpConvertFP(WR[ws]16i+15+WRLEN/2..16i+WRLEN/2, 16)
      WR[wd]32i+31..32i ← f
   endfor

FEXUPL.D
   for i in 0 .. WRLEN/64-1
      f ← UpConvertFP(WR[ws]32i+31+WRLEN/2..32i+WRLEN/2, 32)
      WR[wd]64i+63..64i ← f
   endfor

function UpConvertFP(tt, n)
   /* Implementation defined format up-conversion. */
endfunction UpConvertFP
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception, MSA Floating Point Exception.




**Format:** `FEXUPR.df`

`FEXUPR.W wd,ws` **MSA**

`FEXUPR.D wd,ws` **MSA**

**Purpose:** Vector Floating-Point Up-Convert Interchange Format Right

Vector right elements conversion to wider interchange format.

**Description:** `wd[i] ← up_convert(right_half(ws)[i])`

The right half floating-point elements in vector *ws* are up-converted to a larger interchange format, i.e. from 16-bit to 32-bit, or from 32-bit to 64-bit. The result is written to vector *wd*.

The format up-conversion operation is defined by the IEEE Standard for Floating-Point Arithmetic 754™-2008.

16-bit floating-point inputs are not affected by the flush-to-zero bit FS in MSA Control and Status Register *MSACSR*.

The operands are values in floating-point data format half the size of *df*. The results are floating-point values in data format *df*.

**Restrictions:**

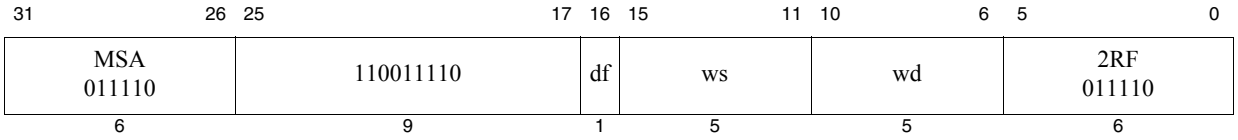
Data-dependent exceptions are possible as specified by the IEEE Standard for Floating-Point Arithmetic 754™-2008.

**Operation:**

```
FEXUPR.W
   for i in 0 .. WRLEN/32-1
      f ← UpConvertFP(WR[ws]16i+15..16i, 16)
      WR[wd]32i+31..32i ← f
   endfor

FEXUPR.D
   for i in 0 .. WRLEN/64-1
      f ← UpConvertFP(WR[ws]32i+31..32i, 32)
      WR[wd]64i+63..64i ← f
   endfor

function UpConvertFP(tt, n)
   /* Implementation defined format up-conversion. */
endfunction UpConvertFP
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception, MSA Floating Point Exception.




**Format:** `FFINT_S.df`

`FFINT_S.W wd,ws` **MSA**

`FFINT_S.D wd,ws` **MSA**

**Purpose:** Vector Floating-Point Round and Convert from Signed Integer

Vector floating-point round and convert from signed integer.

**Description:** `wd[i] ← from_int_s(ws[i])`

The signed integer elements in *ws* are converted to floating-point values. The result is written to vector *wd*.

The integer to floating-point conversion operation is defined by the IEEE Standard for Floating-Point Arithmetic 754™-2008.

The operands are values in integer data format *df*. The results are values in floating-point data format *df*.

**Restrictions:**

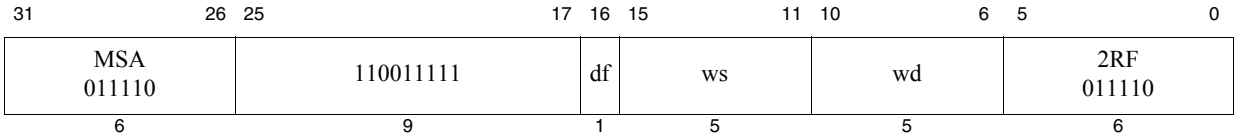
Data-dependent exceptions are possible as specified by the IEEE Standard for Floating-Point Arithmetic 754™-2008.

**Operation:**

```
FFINT_S.W
   for i in 0 .. WRLEN/32-1
      f ← FromIntSignedFP(WR[ws]32i+31..32i, 32)
      WR[wd]32i+31..32i ← f
   endfor

FFINT_S.D
   for i in 0 .. WRLEN/64-1
      f ← FromIntSignedFP(WR[ws]64i+63..64i, 64)
      WR[wd]64i+63..64i ← f
   endfor

function FromFixPointFP(tt, n)
   /* Implementation defined signed integer to floating-point
         conversion. */
endfunction FromFixPointFP
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception, MSA Floating Point Exception.




**Format:** `FFINT_U.df`

`FFINT_U.W wd,ws` **MSA**

`FFINT_U.D wd,ws` **MSA**

**Purpose:** Vector Floating-Point Convert from Unsigned Integer

Vector floating-point convert from unsigned integer.

**Description:** `wd[i] ← from_int_u(ws[i])`

The unsigned integer elements in *ws* are converted to floating-point values. The result is written to vector *wd*.

The integer to floating-point conversion operation is defined by the IEEE Standard for Floating-Point Arithmetic 754™-2008.

The operands are values in integer data format *df*. The results are values in floating-point data format *df*.

**Restrictions:**

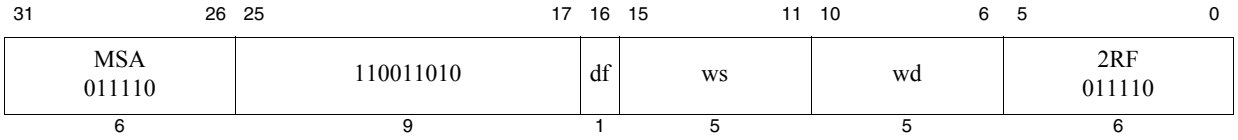
Data-dependent exceptions are possible as specified by the IEEE Standard for Floating-Point Arithmetic 754™-2008.

**Operation:**

```
FFINT_U.W
   for i in 0 .. WRLEN/32-1
      f ← FromIntUnsignedFP(WR[ws]32i+31..32i, 32)
      WR[wd]32i+31..32i ← f
   endfor

FFINT_U.D
   for i in 0 .. WRLEN/64-1
      f ← FromIntUnsignedFP(WR[ws]64i+63..64i, 64)
      WR[wd]64i+63..64i ← f
   endfor

function FromIntUnsignedFP(tt, n)
   /* Implementation defined unsigned integer to floating-point
         conversion. */
endfunction FromIntUnsignedFP
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception, MSA Floating Point Exception.




**Format:** `FFQL.df`

`FFQL.W wd,ws` **MSA**

`FFQL.D wd,ws` **MSA**

**Purpose:** Vector Floating-Point Convert from Fixed-Point Left

Vector left fix-point elements format conversion to floating-point doubling the element width.

**Description:** `wd[i] ← from_q(left_half(ws)[i])`

The left half fixed-point elements in vector *ws* are up-converted to floating-point data format, i.e. from 16-bit Q15 to 32-bit floating-point, or from 32-bit Q31 to 64-bit floating-point. The result is written to vector *wd*.

The fixed-point Q15 or Q31 value is first converted to floating-point as a 16-bit or 32-bit integer (as though it was scaled up by $2^{15}$ or $2^{31}$) and then the resulting floating-point value is scaled down (divided by $2^{15}$ or $2^{31}$).

The scaling and integer to floating-point conversion operations are defined by the IEEE Standard for Floating-Point Arithmetic 754™-2008. No floating-point exceptions are possible because the input data is half the size of the output.

The operands are values in fixed-point data format half the size of *df*. The results are floating-point values in data format *df*.

**Restrictions:**

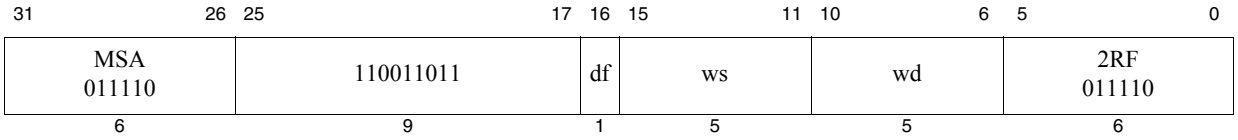
No data-dependent exceptions are possible.

**Operation:**

```
FFQL.W
   for i in 0 .. WRLEN/32-1
      f ← FromFixPointFP(WR[ws]16i+15+WRLEN/2..16i+WRLEN/2, 16)
      WR[wd]32i+31..32i ← f
   endfor

FFQL.D
   for i in 0 .. WRLEN/64-1
      f ← FromFixPointFP(WR[ws]32i+31+WRLEN/2..32i+WRLEN/2, 32)
      WR[wd]64i+63..64i ← f
   endfor

function FromFixPointFP(tt, n)
   /* Implementation defined fixed-point to floating-point conversion. */
endfunction FromFixPointFP
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception.




**Format:** `FFQR.df`

`FFQR.W wd,ws` **MSA**

`FFQR.D wd,ws` **MSA**

**Purpose:** Vector Floating-Point Convert from Fixed-Point Right

Vector right fix-point elements format conversion to floating-point doubling the element width.

**Description:** `wd[i] ← from_q(right_half(ws)[i]);`

The right half fixed-point elements in vector *ws* are up-converted to floating-point data format, i.e. from 16-bit Q15 to 32-bit floating-point, or from 32-bit Q31 to 64-bit floating-point. The result is written to vector *wd*.

The fixed-point Q15 or Q31 value is first converted to floating-point as a 16-bit or 32-bit integer (as though it was scaled up by $2^{15}$ or $2^{31}$) and then the resulting floating-point value is scaled down (divided by $2^{15}$ or $2^{31}$).

The scaling and integer to floating-point conversion operations are defined by the IEEE Standard for Floating-Point Arithmetic 754™-2008. No floating-point exceptions are possible because the input data is half the size of the output.

The operands are values in fixed-point data format half the size of *df*. The results are floating-point values in data format *df*.

**Restrictions:**

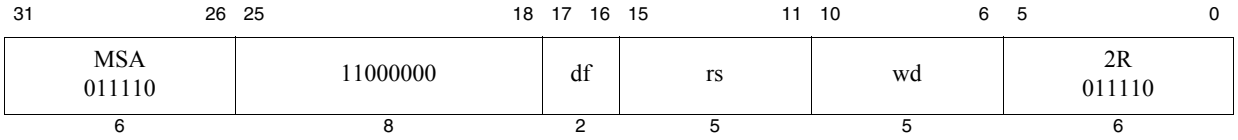
No data-dependent exceptions are possible.

**Operation:**

```
FFQR.W
   for i in 0 .. WRLEN/32-1
      f ← FromFixPointFP(WR[ws]16i+15..16i, 16)
      WR[wd]32i+31..32i ← f
   endfor

FFQR.D
   for i in 0 .. WRLEN/64-1
      f ← FromFixPointFP(WR[wt]32i+31..32i, 32)
      WR[ws]64i+63..64i ← f
   endfor

function FromFixPointFP(tt, n)
   /* Implementation defined fixed-point to floating-point conversion. */
endfunction FromFixPointFP
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception.




**Format:** `FILL.df`

`FILL.B wd,rs` **MSA**

`FILL.H wd,rs` **MSA**

`FILL.W wd,rs` **MSA**

`FILL.D wd,rs` **MIPS64 MSA**

**Purpose:** Vector Fill from GPR

Vector elements replicated from GPR.

**Description:** `wd[i] ← rs`

Replicate GPR *rs* value to all elements in vector *wd*. If the source GPR is wider than the destination data format, the destination's elements will be set to the least significant bits of the GPR.

**Restrictions:**

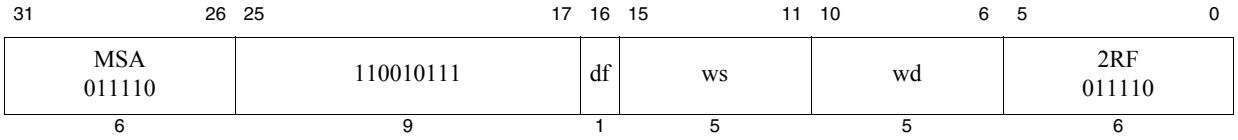
No data-dependent exceptions are possible.

**Operation:**

```
FILL.B
    for i in 0 .. WRLEN/8-1
        WR[wd]8i+7..8i ← GPR[rs]7..0
    endfor

FILL.H
    for i in 0 .. WRLEN/16-1
        WR[wd]16i+15..16i ← GPR[rs]15..0
    endfor

FILL.W
    for i in 0 .. WRLEN/32-1
        WR[wd]32i+31..32i ← GPR[rs]31..0
    endfor

FILL.D
    for i in 0 .. WRLEN/64-1
        WR[wd]64i+63..64i ← GPR[rs]63..0
    endfor
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception.




**Format:** `FLOG2.df`

`FLOG2.W wd,ws` **MSA**

`FLOG2.D wd,ws` **MSA**

**Purpose:** Vector Floating-Point Base 2 Logarithm

Vector floating-point base 2 logarithm.

**Description:** `wd[i] ← log2(ws[i])`

The signed integral base 2 exponents of floating-point elements in vector *ws* are written as floating-point values to vector elements *wd*.

This operation is the homogeneous base 2 **logB()** as defined by the IEEE Standard for Floating-Point Arithmetic 754™-2008.

The *ws* operands and *wd* results are values in floating-point data format *df*.

**Restrictions:**

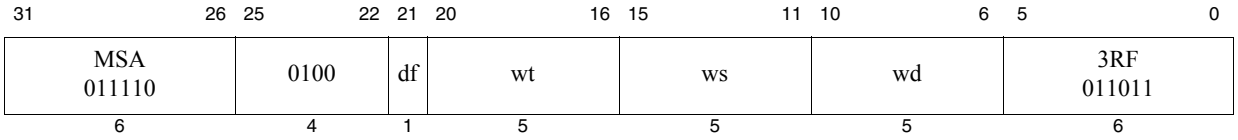
Data-dependent exceptions are possible as specified by the IEEE Standard for Floating-Point Arithmetic 754™-2008.

**Operation:**

```
FLOG2.W
   for i in 0 .. WRLEN/32-1
      l ← Log2FP(WR[ws]32i+31..32i, 32)
      WR[wd]32i+31..32i ← l
   endfor

FLOG2.D
   for i in 0 .. WRLEN/64-1
      f ← Log2FP(WR[ws]64i+63..64i, 64)
      WR[wd]64i+63..64i ← f
   endfor

function Log2FP(tt, n)
   /* Implementation defined logarithm base 2 operation. */
endfunction Log2FP
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception, MSA Floating Point Exception.




**Format:** `FMADD.df`

`FMADD.W wd,ws,wt` **MSA**

`FMADD.D wd,ws,wt` **MSA**

**Purpose:** Vector Floating-Point Multiply-Add

Vector floating-point multiply-add

**Description:** `wd[i] ← wd[i] + ws[i] * wt[i]`

The floating-point elements in vector *wt* multiplied by floating-point elements in vector *ws* are added to the floating-point elements in vector *wd*. The operation is fused, i.e. computed as if with unbounded range and precision, rounding only once to the destination format.

The multiply add operation is defined by the IEEE Standard for Floating-Point Arithmetic 754™-2008. The multiplication between an infinity and a zero signals Invalid Operation exception. If the Invalid Operation exception is disabled, the result is the default quiet NaN.

The operands and results are values in floating-point data format *df*.

**Restrictions:**

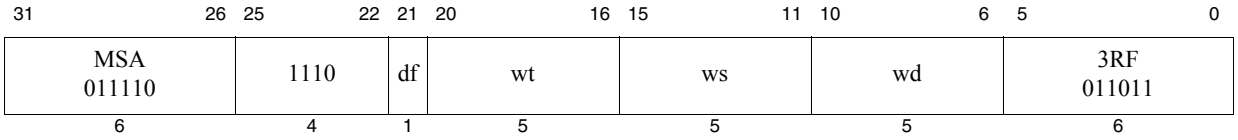
Data-dependent exceptions are possible as specified by the IEEE Standard for Floating-Point Arithmetic 754™-2008.

**Operation:**

```
FMADD.W
    for i in 0 .. WRLEN/32-1
        WR[wd]32i+31..32i ←
                MultiplyAddFP(WR[wd]32i+31..32i, WR[ws]32i+31..32i, WR[wt]32i+31..32i, 32)
    endfor

FMADD.D
    for i in 0 .. WRLEN/64-1
        WR[wd]64i+63..64i ←
                MultiplyAddFP(WR[wd]64i+63..64i, WR[ws]64i+63..64i, WR[wt]64i+63..64i, 64)
        endfor

function MultiplyAddFP(td, tt, ts, n)
    /* Implementation defined multiply add operation. */
endfunction MultiplyAddFP
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception, MSA Floating Point Exception.




**Format:** `FMAX.df`

`FMAX.W wd,ws,wt` **MSA**

`FMAX.D wd,ws,wt` **MSA**

**Purpose:** Vector Floating-Point Maximum

Vector floating-point maximum.

**Description:** `wd[i] ← max(ws[i], wt[i])`

The largest values between corresponding floating-point elements in vector *ws* and vector *wt* are written to vector *wd*.

The largest value is defined by the maxNum operation in the IEEE Standard for Floating-Point Arithmetic 754™-2008.

The operands and results are values in floating-point data format *df*.

**Restrictions:**

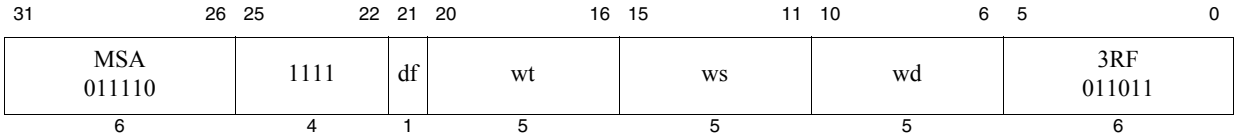
Data-dependent exceptions are possible as specified by the IEEE Standard for Floating-Point Arithmetic 754™-2008.

**Operation:**

```
FMAX.W
   for i in 0 .. WRLEN/32-1
      WR[wd]32i+31..32i ← MaxFP(WR[ws]32i+31..32i, WR[wt]32i+31..32i, 32)
   endfor

FMAX.D
   for i in 0 .. WRLEN/64-1
      WR[wd]64i+63..64i ← MaxFP(WR[ws]64i+63..64i, WR[wt]64i+63..64i, 64)
   endfor

function MaxFP(tt, ts, n)
   /* Implementation defined, returns the largest argument. */
endfunction MaxFP
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception, MSA Floating Point Exception.




**Format:** `FMAX_A.df`

`FMAX_A.W wd,ws,wt` **MSA**

`FMAX_A.D wd,ws,wt` **MSA**

**Purpose:** Vector Floating-Point Maximum Based on Absolute Values

Vector floating-point maximum based on the magnitude, i.e. absolute values.

**Description:** `wd[i] ← absolute_value(ws[i]) > absolute_value(wt[i])? ws[i]: wt[i]`

The value with the largest magnitude, i.e. absolute value, between corresponding floating-point elements in vector *ws* and vector *wt* are written to vector *wd*.

The largest absolute value is defined by the maxNumMag operation in the IEEE Standard for Floating-Point Arithmetic 754™-2008.

The operands and results are values in floating-point data format *df*.

**Restrictions:**

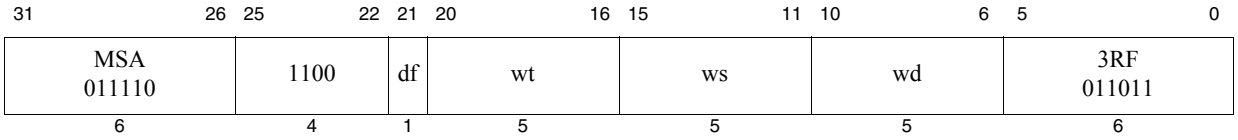
Data-dependent exceptions are possible as specified by the IEEE Standard for Floating-Point Arithmetic 754™-2008.

**Operation:**

```
FMAX_A.W
   for i in 0 .. WRLEN/32-1
      WR[wd]32i+31..32i ← MaxAbsoluteFP(WR[ws]32i+31..32i, WR[wt]32i+31..32i, 32)
   endfor

FMAX_A.D
   for i in 0 .. WRLEN/64-1
      WR[wd]64i+63..64i ← MaxAbsoluteFP(WR[ws]64i+63..64i, WR[wt]64i+63..64i, 64)
   endfor

function MaxAbsoluteFP(tt, ts, n)
   /* Implementation defined, returns the argument with largest
         absolute value. For equal absolute values, returns the largest
         argument.*/
endfunction MaxAbsoluteFP
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception, MSA Floating Point Exception.




**Format:** `FMIN.df`

`FMIN.W wd,ws,wt` **MSA**

`FMIN.D wd,ws,wt` **MSA**

**Purpose:** Vector Floating-Point Minimum

Vector floating-point minimum.

**Description:** `wd[i] ← min(ws[i], wt[i])`

The smallest value between corresponding floating-point elements in vector *ws* and vector *wt* are written to vector *wd*.

The smallest value is defined by the minNum operation in the IEEE Standard for Floating-Point Arithmetic 754™-2008.

The operands and results are values in floating-point data format *df*.

**Restrictions:**

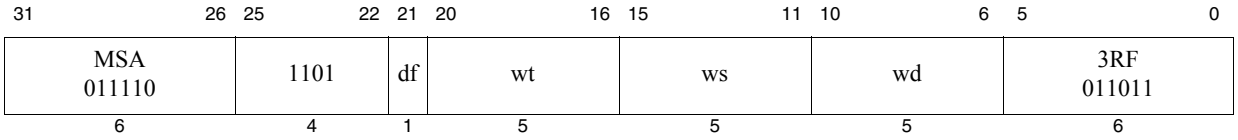
Data-dependent exceptions are possible as specified by the IEEE Standard for Floating-Point Arithmetic 754™-2008.

**Operation:**

```
FMIN.W
   for i in 0 .. WRLEN/32-1
      WR[wd]32i+31..32i ← MinFP(WR[ws]32i+31..32i, WR[wt]32i+31..32i, 32)
   endfor

FMIN.D
   for i in 0 .. WRLEN/64-1
      WR[wd]64i+63..64i ← MinFP(WR[ws]64i+63..64i, WR[wt]64i+63..64i, 64)
   endfor

function MinFP(tt, ts, n)
   /* Implementation defined, returns the smallest argument. */
endfunction MinFP
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception, MSA Floating Point Exception.




**Format:** `FMIN_A.df`

`FMIN_A.W wd,ws,wt` **MSA**

`FMIN_A.D wd,ws,wt` **MSA**

**Purpose:** Vector Floating-Point Minimum Based on Absolute Values

Vector floating-point minimum based on the magnitude, i.e. absolute values.

**Description:** `wd[i] ← absolute_value(ws[i]) < absolute_value(wt[i])? ws[i]: wt[i]`

The value with the smallest magnitude, i.e. absolute value, between corresponding floating-point elements in vector *ws* and vector *wt* are written to vector *wd*.

The smallest absolute value is defined by the minNumMag operation in the IEEE Standard for Floating-Point Arithmetic 754™-2008.

The operands and results are values in floating-point data format *df*.

**Restrictions:**

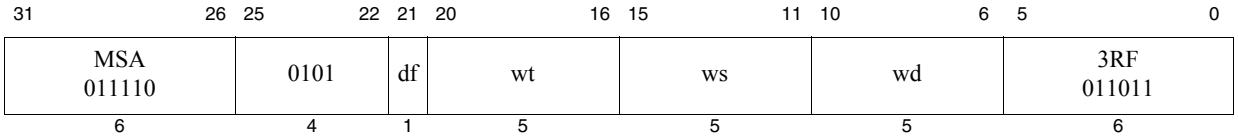
Data-dependent exceptions are possible as specified by the IEEE Standard for Floating-Point Arithmetic 754™-2008.

**Operation:**

```
FMIN_A.W
   for i in 0 .. WRLEN/32-1
      WR[wd]32i+31..32i ← MinAbsoluteFP(WR[ws]32i+31..32i, WR[wt]32i+31..32i, 32)
   endfor

FMIN_A.D
   for i in 0 .. WRLEN/64-1
      WR[wd]64i+63..64i ← MinAbsoluteFP(WR[ws]64i+63..64i, WR[wt]64i+63..64i, 64)
   endfor

function MinAbsoluteFP(tt, ts, n)
   /* Implementation defined, returns the argument with smallest
         absolute value. For equal absolute values, returns the smallest
         argument.*/
endfunction MinAbsoluteFP
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception, MSA Floating Point Exception.




**Format:** `FMSUB.df`

`FMSUB.W wd,ws,wt` **MSA**

`FMSUB.D wd,ws,wt` **MSA**

**Purpose:** Vector Floating-Point Multiply-Sub

Vector floating-point multiply-sub

**Description:** `wd[i] ← wd[i] - ws[i] * wt[i]`

The floating-point elements in vector *wt* multiplied by floating-point elements in vector *ws* are subtracted from the floating-point elements in vector *wd*. The operation is fused, i.e. computed as if with unbounded range and precision, rounding only once to the destination format.

The multiply subtract operation is defined by the IEEE Standard for Floating-Point Arithmetic 754™-2008. The multiplication between an infinity and a zero signals Invalid Operation exception. If the Invalid Operation exception is disabled, the result is the default quiet NaN.

The operands and results are values in floating-point data format *df*.

**Restrictions:**

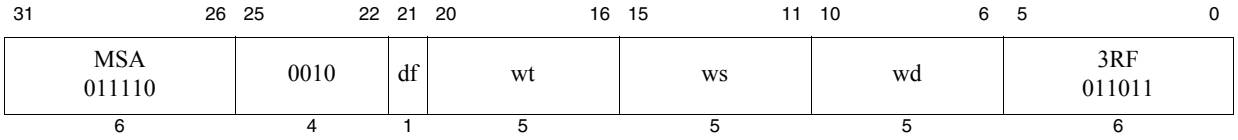
Data-dependent exceptions are possible as specified by the IEEE Standard for Floating-Point Arithmetic 754™-2008.

**Operation:**

```
FMSUB.W
   for i in 0 .. WRLEN/32-1
      WR[wd]32i+31..32i ←
            MultiplySubFP(WR[wd]32i+31..32i, WR[ws]32i+31..32i, WR[wt]32i+31..32i, 32)
   endfor

FMSUB.D
   for i in 0 .. WRLEN/64-1
      WR[wd]64i+63..64i ←
            MultiplySubFP(WR[wd]64i+63..64i, WR[ws]64i+63..64i, WR[wt]64i+63..64i, 64)
   endfor

function MultiplySubFP(td, tt, ts, n)
   /* Implementation defined multiply subtract operation. */
endfunction MultiplySubFP
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception, MSA Floating Point Exception.




**Format:** `FMUL.df`

`FMUL.W wd,ws,wt` **MSA**

`FMUL.D wd,ws,wt` **MSA**

**Purpose:** Vector Floating-Point Multiplication

Vector floating-point multiplication.

**Description:** `wd[i] ← ws[i] * wt[i]`

The floating-point elements in vector *wt* are multiplied by the floating-point elements in vector *ws*. The result is written to vector *wd*.

The multiplication operation is defined by the IEEE Standard for Floating-Point Arithmetic 754™-2008.

The operands and results are values in floating-point data format *df*.

**Restrictions:**

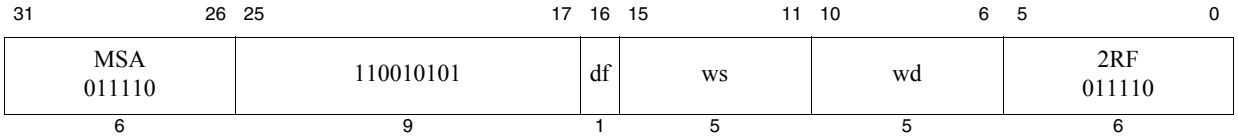
Data-dependent exceptions are possible as specified by the IEEE Standard for Floating-Point Arithmetic 754™-2008.

**Operation:**

```
FMUL.W
    for i in 0 .. WRLEN/32-1
        WR[wd]32i+31..32i ← MultiplyFP(WR[ws]32i+31..32i, WR[wt]32i+31..32i, 32)
    endfor

FMUL.D
    for i in 0 .. WRLEN/64-1
        WR[wd]64i+63..64i ← MultiplyFP(WR[ws]64i+63..64i, WR[wt]64i+63..64i, 64)
    endfor

function MultiplyFP(tt, ts, n)
    /* Implementation defined multiplication operation. */
endfunction MultiplyFP
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception, MSA Floating Point Exception.




**Format:** `FRCP.df`

`FRCP.W wd,ws` **MSA**

`FRCP.D wd,ws` **MSA**

**Purpose:** Vector Approximate Floating-Point Reciprocal

Vector floating-point reciprocal.

**Description:** `wd[i] ← 1.0 / ws[i]`

The reciprocals of floating-point elements in vector *ws* are calculated as specified below. The result is written to vector *wd*.

The compliant reciprocal operation is defined as 1.0 divided by element value, where the IEEE Standard for Floating-Point Arithmetic 754™-2008 defined divide operation is affected by the rounding mode bits RM and flush-to-zero bit FS in MSA Control and Status Register *MSACSR*. The compliant reciprocals signal all the exceptions specified by the IEEE Standard for Floating-Point Arithmetic 754™-2008 for the divide operation.

The reciprocal operation is allowed to be approximate. The approximation differs from the compliant reciprocal representation by no more than one unit in the least significant place. Approximate reciprocal operations signal the Inexact exception if the compliant reciprocal is Inexact or if there is a chance the approximated result may differ from the compliant reciprocal. Approximate reciprocal operations are allowed to not signal the Overflow or Underflow exceptions. The Invalid and divide by Zero exceptions are signaled based on the IEEE Standard for Floating-Point Arithmetic 754™-2008 defined divide operation.

The operands and results are values in floating-point data format *df*.

**Restrictions:**

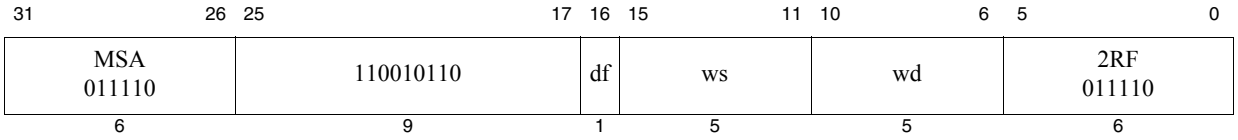
Data-dependent exceptions are possible as specified by the IEEE Standard for Floating-Point Arithmetic 754™-2008.

**Operation:**

```
FRCP.W
   for i in 0 .. WRLEN/32-1
      WR[wd]32i+31..32i ← ReciprocalFP(WR[ws]32i+31..32i, 32)
   endfor

FRCP.D
   for i in 0 .. WRLEN/64-1
      WR[wd]64i+63..64i ← ReciprocalFP(WR[ws]64i+63..64i, 64)
   endfor

function ReciprocalFP(tt, ts, n)
   /* Implementation defined Reciprocal operation. */
endfunction ReciprocalFP
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception, MSA Floating Point Exception.




**Format:** `FRINT.df`

`FRINT.W wd,ws` **MSA**

`FRINT.D wd,ws` **MSA**

**Purpose:** Vector Floating-Point Round to Integer

Vector floating-point round to integer.

**Description:** `wd[i] ← round_int(ws[i])`

The floating-point elements in vector *ws* are rounded to an integral valued floating-point number in the same format based on the rounding mode bits RM in MSA Control and Status Register *MSACSR*. The result is written to vector *wd*.

The round to integer operation is exact as defined by the IEEE Standard for Floating-Point Arithmetic 754™-2008, i.e. the Inexact exception is signaled if the result does not have the same numerical value as the input operand.

The operands and results are values in floating-point data format *df*.

**Restrictions:**

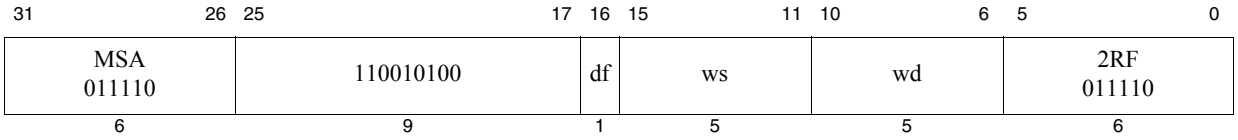
Data-dependent exceptions are possible as specified by the IEEE Standard for Floating-Point Arithmetic 754™-2008.

**Operation:**

```
FRINT.W
   for i in 0 .. WRLEN/32-1
      f ← RoundIntFP(WR[ws]32i+31..32i, 32)
      WR[wd]32i+31..32i ← f
   endfor

FRINT.D
   for i in 0 .. WRLEN/64-1
      f ← RoundIntFP(WR[ws]64i+63..64i, 64)
      WR[wd]64i+63..64i ← f
   endfor

function RoundIntFP(tt, n)
   /* Implementation defined round to integer operation. */
endfunction RoundIntFP
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception, MSA Floating Point Exception.




**Format:** `FRSQRT.df`

`FRSQRT.W wd,ws` **MSA**

`FRSQRT.D wd,ws` **MSA**

**Purpose:** Vector Approximate Floating-Point Reciprocal of Square Root

Vector floating-point reciprocal of square root.

**Description:** `wd[i] ← 1.0 / sqrt(ws[i])`

The reciprocals of the square roots of floating-point elements in vector *ws* are calculated as specified below. The result is written to vector *wd*.

The compliant reciprocal of the square root operation is defined as 1.0 divided by the square root of the element value, where the IEEE Standard for Floating-Point Arithmetic 754™-2008 defined divide and square root operations are affected by the rounding mode bits RM and flush-to-zero bit FS in MSA Control and Status Register *MSACSR*. The compliant reciprocals of the square roots signal all the exceptions specified by the IEEE Standard for Floating-Point Arithmetic 754™-2008 for the divide and square roots operations.

The reciprocal of the square root operation is allowed to be approximate. The approximation differs from the compliant reciprocal of the square root representation by no more than two units in the least significant place. Approximate reciprocal of the square root operations signal the Inexact exception if the compliant reciprocal of the square root is Inexact or if there is a chance the approximated result may differ from the compliant reciprocal of the square root. The Invalid and divide by Zero exceptions are signaled based on the IEEE Standard for Floating-Point Arithmetic 754™-2008 defined divide operation.

The operands and results are values in floating-point data format *df*.

**Restrictions:**

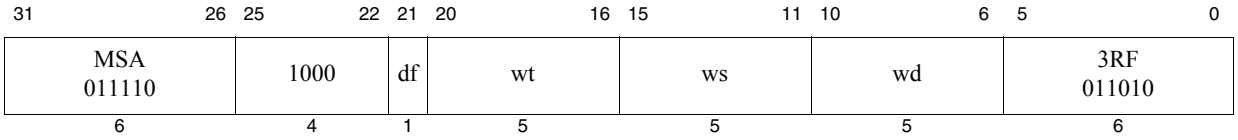
Data-dependent exceptions are possible as specified by the IEEE Standard for Floating-Point Arithmetic 754™-2008.

**Operation:**

```
FRSQRT.W
   for i in 0 .. WRLEN/32-1
      f ← SquareRootReciprocalFP(WR[ws]32i+31..32i, 32)
      WR[wd]32i+31..32i ← f
   endfor

FRSQRT.D
   for i in 0 .. WRLEN/64-1
      f ← SquareRootReciprocalFP(WR[ws]64i+63..64i, 64)
      WR[wd]64i+63..64i ← f
   endfor

function SquareRootReciprocalFP(tt, ts, n)
   /* Implementation defined square root reciprocal operation. */
endfunction SquareRootReciprocalFP
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception, MSA Floating Point Exception.




**Format:** `FSAF.df`

`FSAF.W wd,ws,wt` **MSA**

`FSAF.D wd,ws,wt` **MSA**

**Purpose:** Vector Floating-Point Signaling Compare Always False

Vector to vector floating-point signaling compare always false; all destination bits are clear.

**Description:** `wd[i] ← signalingFalse(ws[i], wt[i])`

Set all bits to 0 in *wd* elements. Signaling and quiet NaN elements in *ws* or *wt* signal Invalid Operation exception.

The Inexact Exception is not signaled when subnormal input operands are flushed based on the flush-to-zero bit FS in MSA Control and Status Register *MSACSR*. In case of a floating-point exception, the default result has all bits set to 0.

The operands are values in floating-point data format *df*. The results are values in integer data format *df*.

**Restrictions:**

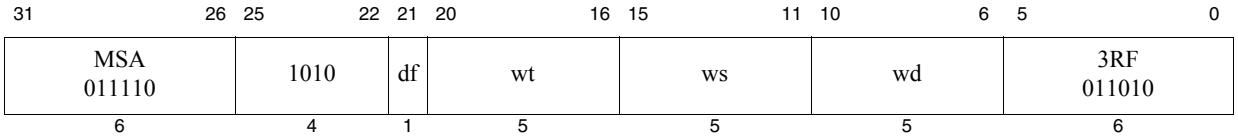
Data-dependent exceptions are possible as specified by the IEEE Standard for Floating-Point Arithmetic 754™-2008.

**Operation:**

```
FSAF.W
   for i in 0 .. WRLEN/32-1
      WR[wd]32i+31..32i ← SignalingFALSE(WR[ws]32i+31..32i, WR[wt]32i+31..32i, 32)
   endfor

FSAF.D
   for i in 0 .. WRLEN/64-1
      WR[wd]64i+63..64i ← SignalingFALSE(WR[ws]64i+63..64i, WR[wt]64i+63..64i, 64)
   endfor

function SignalingFALSE(tt, ts, n)
   /* Implementation defined signaling and quiet NaN test */
   return 0
endfunction SignalingFALSE
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception, MSA Floating Point Exception.




**Format:** `FSEQ.df`

`FSEQ.W wd,ws,wt` **MSA**

`FSEQ.D wd,ws,wt` **MSA**

**Purpose:** Vector Floating-Point Signaling Compare Equal

Vector to vector floating-point signaling compare for equality; if true all destination bits are set, otherwise clear.

**Description:** `wd[i] ← (ws[i] =(signaling) wt[i])`

Set all bits to 1 in *wd* elements if the corresponding *ws* and *wt* floating-point elements are equal, otherwise set all bits to 0.

The signaling compare operation is defined by the IEEE Standard for Floating-Point Arithmetic 754™-2008.

The Inexact Exception is not signaled when subnormal input operands are flushed based on the flush-to-zero bit FS in MSA Control and Status Register *MSACSR*. In case of a floating-point exception, the default result has all bits set to 0.

The operands are values in floating-point data format *df*. The results are values in integer data format *df*.

**Restrictions:**

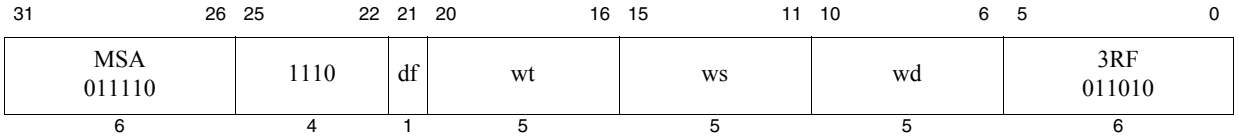
Data-dependent exceptions are possible as specified by the IEEE Standard for Floating-Point Arithmetic 754™-2008.

**Operation:**

```
FSEQ.W
   for i in 0 .. WRLEN/32-1
      c ← EqualSigFP(WR[ws]32i+31..32i, WR[wt]32i+31..32i, 32)
      WR[wd]32i+31..32i ← c32
   endfor

FSEQ.D
   for i in 0 .. WRLEN/64-1
      c ← EqualSigFP(WR[ws]64i+63..64i, WR[wt]64i+63..64i, 64)
      WR[wd]64i+63..64i ← c64
   endfor

function EqualSigFP(tt, ts, n)
   /* Implementation defined signaling equal compare operation. */
endfunction EqualSigFP
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception, MSA Floating Point Exception.




**Format:** `FSLE.df`

`FSLE.W wd,ws,wt` **MSA**

`FSLE.D wd,ws,wt` **MSA**

**Purpose:** Vector Floating-Point Signaling Compare Less or Equal

Vector to vector floating-point signaling compare for less than or equal; if true all destination bits are set, otherwise clear.

**Description:** `wd[i] ← (ws[i] <=(signaling) wt[i])`

Set all bits to 1 in *wd* elements if the corresponding *ws* floating-point elements are less than or equal to *wt* floating-point elements, otherwise set all bits to 0.

The signaling compare operation is defined by the IEEE Standard for Floating-Point Arithmetic 754™-2008.

The Inexact Exception is not signaled when subnormal input operands are flushed based on the flush-to-zero bit FS in MSA Control and Status Register *MSACSR*. In case of a floating-point exception, the default result has all bits set to 0.

The operands are values in floating-point data format *df*. The results are values in integer data format *df*.

**Restrictions:**

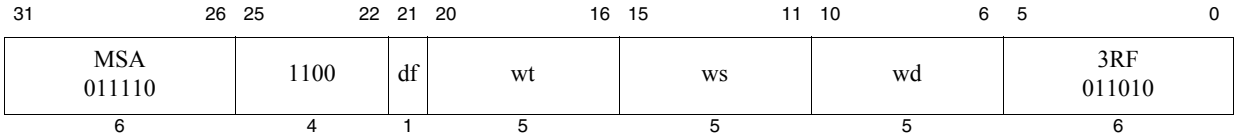
Data-dependent exceptions are possible as specified by the IEEE Standard for Floating-Point Arithmetic 754™-2008.

**Operation:**

```
FSLE.W
    for i in 0 .. WRLEN/32-1
        c ← LessSigFP(WR[ws]32i+31..32i, WR[wt]32i+31..32i, 32)
        d ← EqualSigFP(WR[ws]32i+31..32i, WR[wt]32i+31..32i, 32)
        WR[wd]32i+31..32i ← (c | d)32
    endfor

FSLE.D
    for i in 0 .. WRLEN/64-1
        c ← LessSigFP(WR[ws]64i+63..64i, WR[wt]64i+63..64i, 64)
        d ← EqualSigFP(WR[ws]64i+63..64i, WR[wt]64i+63..64i, 64)
        WR[wd]64i+63..64i ← (c | d)64
    endfor

function LessThanSigFP(tt, ts, n)
    /* Implementation defined signaling less than compare operation. */
endfunction LessThanSigFP

function EqualSigFP(tt, ts, n)
    /* Implementation defined signaling equal compare operation. */
endfunction EqualSigFP
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception, MSA Floating Point Exception.




**Format:** `FSLT.df`

`FSLT.W wd,ws,wt` **MSA**

`FSLT.D wd,ws,wt` **MSA**

**Purpose:** Vector Floating-Point Signaling Compare Less Than

Vector to vector floating-point signaling compare for less than; if true all destination bits are set, otherwise clear.

**Description:** `wd[i] ← (ws[i] <(signaling) wt[i])`

Set all bits to 1 in *wd* elements if the corresponding *ws* floating-point elements are less than *wt* floating-point elements, otherwise set all bits to 0.

The signaling compare operation is defined by the IEEE Standard for Floating-Point Arithmetic 754™-2008.

The Inexact Exception is not signaled when subnormal input operands are flushed based on the flush-to-zero bit FS in MSA Control and Status Register *MSACSR*. In case of a floating-point exception, the default result has all bits set to 0.

The operands are values in floating-point data format *df*. The results are values in integer data format *df*.

**Restrictions:**

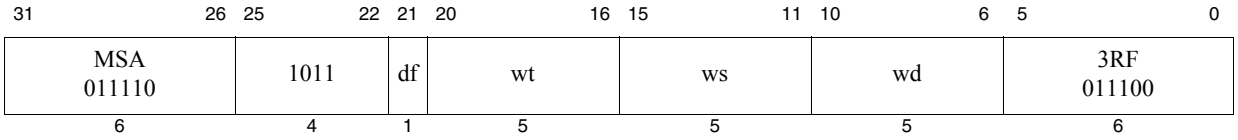
Data-dependent exceptions are possible as specified by the IEEE Standard for Floating-Point Arithmetic 754™-2008.

**Operation:**

```
FSLT.W
    for i in 0 .. WRLEN/32-1
        c ← LessSigFP(WR[ws]32i+31..32i, WR[wt]32i+31..32i, 32)
        WR[wd]32i+31..32i ← c32
    endfor

FSLT.D
    for i in 0 .. WRLEN/64-1
        c ← LessSigFP(WR[ws]64i+63..64i, WR[wt]64i+63..64i, 64)
        WR[wd]64i+63..64i ← c64
    endfor

function LessThanSigFP(tt, ts, n)
    /* Implementation defined signaling less than compare operation. */
endfunction LessThanSigFP
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception, MSA Floating Point Exception.




**Format:** `FSNE.df`

`FSNE.W wd,ws,wt` **MSA**

`FSNE.D wd,ws,wt` **MSA**

**Purpose:** Vector Floating-Point Signaling Compare Not Equal

Vector to vector floating-point signaling compare for not equal; if true all destination bits are set, otherwise clear.

**Description:** `wd[i] ← (ws[i] ≠(signaling) wt[i])`

Set all bits to 1 in *wd* elements if the corresponding *ws* and *wt* floating-point elements are not equal, otherwise set all bits to 0.

The signaling compare operation is defined by the IEEE Standard for Floating-Point Arithmetic 754™-2008.

The Inexact Exception is not signaled when subnormal input operands are flushed based on the flush-to-zero bit FS in MSA Control and Status Register *MSACSR*. In case of a floating-point exception, the default result has all bits set to 0.

The operands are values in floating-point data format *df*. The results are values in integer data format *df*.

**Restrictions:**

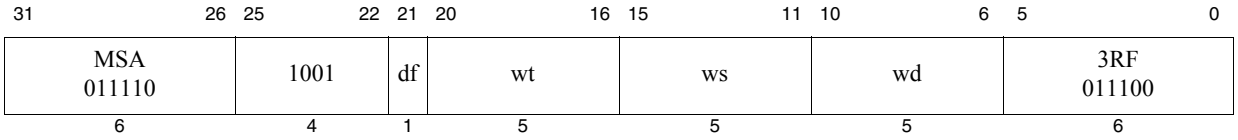
Data-dependent exceptions are possible as specified by the IEEE Standard for Floating-Point Arithmetic 754™-2008.

**Operation:**

```
FSNE.W
    for i in 0 .. WRLEN/32-1
        c ← NotEqualSigFP(WR[ws]32i+31..32i, WR[wt]32i+31..32i, 32)
        WR[wd]32i+31..32i ← c32
    endfor

FSNE.D
    for i in 0 .. WRLEN/64-1
        c ← NotEqualSigFP(WR[ws]64i+63..64i, WR[wt]64i+63..64i, 64)
        WR[wd]64i+63..64i ← c64
    endfor

function NotEqualSigFP(tt, ts, n)
    /* Implementation defined signaling not equal compare operation. */
endfunction NotEqualSigFP
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception, MSA Floating Point Exception.




**Format:** `FSOR.df`

`FSOR.W wd,ws,wt` **MSA**

`FSOR.D wd,ws,wt` **MSA**

**Purpose:** Vector Floating-Point Signaling Compare Ordered

Vector to vector floating-point signaling compare ordered; if true all destination bits are set, otherwise clear.

**Description:** `wd[i] ← ws[i] !?(signaling) wt[i]`

Set all bits to 1 in *wd* elements if the corresponding *ws* and *wt* floating-point elements are ordered, i.e. both elements are not NaN values, otherwise set all bits to 0.

The signaling compare operation is defined by the IEEE Standard for Floating-Point Arithmetic 754™-2008.

The Inexact Exception is not signaled when subnormal input operands are flushed based on the flush-to-zero bit FS in MSA Control and Status Register *MSACSR*. In case of a floating-point exception, the default result has all bits set to 0.

The operands are values in floating-point data format *df*. The results are values in integer data format *df*.

**Restrictions:**

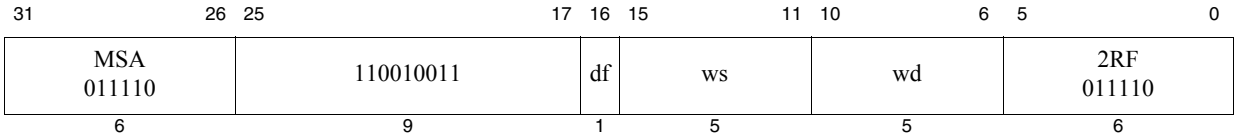
Data-dependent exceptions are possible as specified by the IEEE Standard for Floating-Point Arithmetic 754™-2008.

**Operation:**

```
FSOR.W
    for i in 0 .. WRLEN/32-1
        c ← OrderedSigFP(WR[ws]32i+31..32i, WR[wt]32i+31..32i, 32)
        WR[wd]32i+31..32i ← c32
    endfor

FSOR.D
    for i in 0 .. WRLEN/64-1
        c ← OrderedSigFP(WR[ws]64i+63..64i, WR[wt]64i+63..64i, 64)
        WR[wd]64i+63..64i ← c64
    endfor

function OrderedSigFP(tt, ts, n)
    /* Implementation defined signaling ordered compare operation. */
endfunction OrderedSigFP
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception, MSA Floating Point Exception.




**Format:** `FSQRT.df`

`FSQRT.W wd,ws` **MSA**

`FSQRT.D wd,ws` **MSA**

**Purpose:** Vector Floating-Point Square Root

Vector floating-point square root.

**Description:** `wd[i] ← sqrt(ws[i])`

The square roots of floating-point elements in vector *ws* are written to vector *wd*.

The square root operation is defined by the IEEE Standard for Floating-Point Arithmetic 754™-2008.

The operands and results are values in floating-point data format *df*.

**Restrictions:**

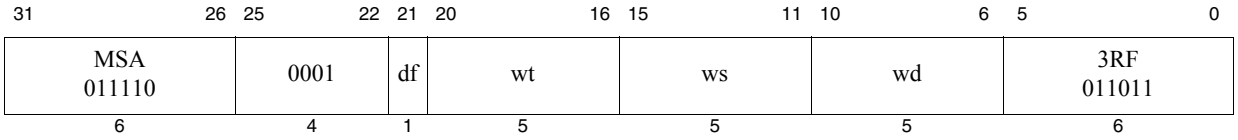
Data-dependent exceptions are possible as specified by the IEEE Standard for Floating-Point Arithmetic 754™-2008.

**Operation:**

```
FSQRT.W
    for i in 0 .. WRLEN/32-1
        f ← SquareRootFP(WR[ws]32i+31..32i, 32)
        WR[wd]32i+31..32i ← f
    endfor

FSQRT.D
    for i in 0 .. WRLEN/64-1
        f ← SquareRootFP(WR[ws]64i+63..64i, 64)
        WR[wd]64i+63..64i ← f
    endfor

function SquareRootFP(tt, ts, n)
    /* Implementation defined square root operation. */
endfunction SquareRootFP
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception, MSA Floating Point Exception.




**Format:** `FSUB.df`

`FSUB.W wd,ws,wt` **MSA**

`FSUB.D wd,ws,wt` **MSA**

**Purpose:** Vector Floating-Point Subtraction

Vector floating-point subtraction.

**Description:** `wd[i] ← ws[i] - wt[i]`

The floating-point elements in vector *wt* are subtracted from the floating-point elements in vector *ws*. The result is written to vector *wd*.

The subtract operation is defined by the IEEE Standard for Floating-Point Arithmetic 754™-2008.

The operands and results are values in floating-point data format *df*.

**Restrictions:**

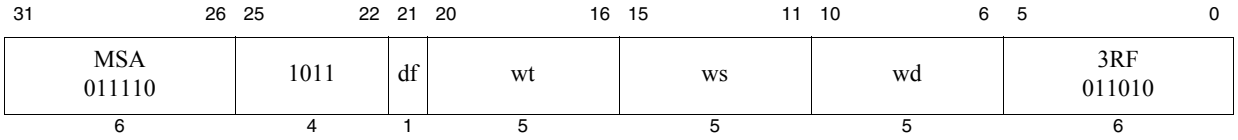
Data-dependent exceptions are possible as specified by the IEEE Standard for Floating-Point Arithmetic 754™-2008.

**Operation:**

```
FSUB.W
   for i in 0 .. WRLEN/32-1
      WR[wd]32i+31..32i ← SubtractFP(WR[ws]32i+31..32i, WR[wt]32i+31..32i, 32)
   endfor

FSUB.D
   for i in 0 .. WRLEN/64-1
      WR[wd]64i+63..64i ← SubtractFP(WR[ws]64i+63..64i, WR[wt]64i+63..64i, 64)
   endfor

function SubtractFP(tt, ts, n)
   /* Implementation defined subtract operation. */
endfunction SubtractFP
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception, MSA Floating Point Exception.




**Format:** `FSUEQ.df`

`FSUEQ.W wd,ws,wt` **MSA**

`FSUEQ.D wd,ws,wt` **MSA**

**Purpose:** Vector Floating-Point Signaling Compare Unordered or Equal

Vector to vector floating-point signaling compare for unordered or equality; if true all destination bits are set, otherwise clear.

**Description:** `wd[i] ← (ws[i] =?(signaling) wt[i])`

Set all bits to 1 in *wd* elements if the corresponding *ws* and *wt* floating-point elements are unordered or equal, otherwise set all bits to 0.

The signaling compare operation is defined by the IEEE Standard for Floating-Point Arithmetic 754™-2008.

The Inexact Exception is not signaled when subnormal input operands are flushed based on the flush-to-zero bit FS in MSA Control and Status Register *MSACSR*. In case of a floating-point exception, the default result has all bits set to 1.

The operands are values in floating-point data format *df*. The results are values in integer data format *df*.

**Restrictions:**

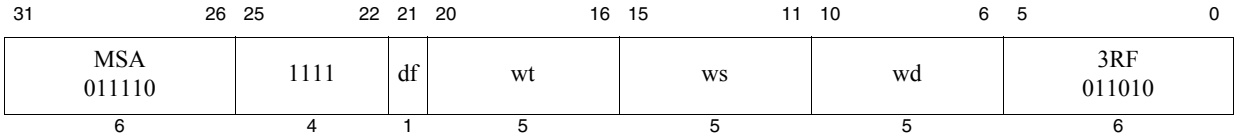
Data-dependent exceptions are possible as specified by the IEEE Standard for Floating-Point Arithmetic 754™-2008.

**Operation:**

```
FSUEQ.W
    for i in 0 .. WRLEN/32-1
        c ← UnorderedSigFP(WR[ws]32i+31..32i, WR[wt]32i+31..32i, 32)
        d ← EqualSigFP(WR[ws]32i+31..32i, WR[wt]32i+31..32i, 32)
        WR[wd]32i+31..32i ← (c | d)32
    endfor

FSUEQ.D
    for i in 0 .. WRLEN/64-1
        c ← UnorderedSigFP(WR[ws]64i+63..64i, WR[wt]64i+63..64i, 64)
        d ← EqualSigFP(WR[ws]64i+63..64i, WR[wt]64i+63..64i, 64)
        WR[wd]64i+63..64i ← (c | d)64
    endfor

function UnorderedSigFP(tt, ts, n)
    /* Implementation defined signaling unordered compare operation. */
endfunction UnorderedSigFP

function EqualSigFP(tt, ts, n)
    /* Implementation defined signaling equal compare operation. */
endfunction EqualSigFP
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception, MSA Floating Point Exception.




**Format:** `FSULE.df`

`FSULE.W wd,ws,wt` **MSA**

`FSULE.D wd,ws,wt` **MSA**

**Purpose:** Vector Floating-Point Signaling Compare Unordered or Less or Equal

Vector to vector floating-point signaling compare for unordered or less than or equal; if true all destination bits are set, otherwise clear.

**Description:** `wd[i] ← (ws[i] <=?(signaling) wt[i])`

Set all bits to 1 in *wd* elements if the corresponding *ws* floating-point elements are unordered or less than or equal to *wt* floating-point elements, otherwise set all bits to 0.

The signaling compare operation is defined by the IEEE Standard for Floating-Point Arithmetic 754™-2008.

The Inexact Exception is not signaled when subnormal input operands are flushed based on the flush-to-zero bit FS in MSA Control and Status Register *MSACSR*. In case of a floating-point exception, the default result has all bits set to 1.

The operands are values in floating-point data format *df*. The results are values in integer data format *df*.

**Restrictions:**

Data-dependent exceptions are possible as specified by the IEEE Standard for Floating-Point Arithmetic 754™-2008.

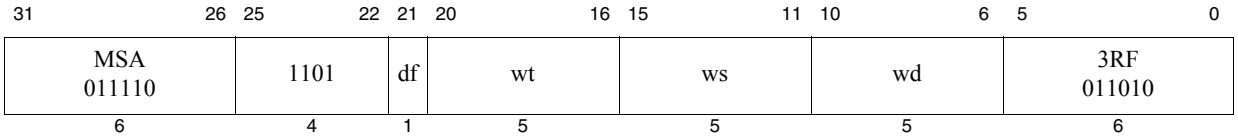
**Operation:**

```
FSULE.W
   for i in 0 .. WRLEN/32-1
      c ← UnorderedSigFP(WR[ws]32i+31..32i, WR[wt]32i+31..32i, 32)
      d ← LessSigFP(WR[ws]32i+31..32i, WR[wt]32i+31..32i, 32)
      e ← EqualSigFP(WR[ws]32i+31..32i, WR[wt]32i+31..32i, 32)
      WR[wd]32i+31..32i ← (c | d | e)32
   endfor

FSULE.D
   for i in 0 .. WRLEN/64-1
      c ← UnorderedSigFP(WR[ws]64i+63..64i, WR[wt]64i+63..64i, 64)
      d ← LessSigFP(WR[ws]64i+63..64i, WR[wt]64i+63..64i, 64)
      e ← EqualSigFP(WR[ws]64i+63..64i, WR[wt]64i+63..64i, 64)
      WR[wd]64i+63..64i ← (c | d | e)64
   endfor

function UnorderedSigFP(tt, ts, n)
   /* Implementation defined signaling unordered compare operation. */
endfunction UnorderedSigFP

function LessThanSigFP(tt, ts, n)
   /* Implementation defined signaling less than compare operation. */
endfunction LessThanSigFP
```

```
function EqualSigFP(tt, ts, n)
   /* Implementation defined signaling equal compare operation. */
endfunction EqualSigFP
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception, MSA Floating Point Exception.




**Format:** `FSULT.df`

`FSULT.W wd,ws,wt` **MSA**

`FSULT.D wd,ws,wt` **MSA**

**Purpose:** Vector Floating-Point Signaling Compare Unordered or Less Than

Vector to vector floating-point signaling compare for unordered or less than; if true all destination bits are set, otherwise clear.

**Description:** `wd[i] ← (ws[i] <?(signaling) wt[i])`

Set all bits to 1 in *wd* elements if the corresponding *ws* floating-point elements are unordered or less than *wt* floating-point elements, otherwise set all bits to 0.

The signaling compare operation is defined by the IEEE Standard for Floating-Point Arithmetic 754™-2008.

The Inexact Exception is not signaled when subnormal input operands are flushed based on the flush-to-zero bit FS in MSA Control and Status Register *MSACSR*. In case of a floating-point exception, the default result has all bits set to 1.

The operands are values in floating-point data format *df*. The results are values in integer data format *df*.

**Restrictions:**

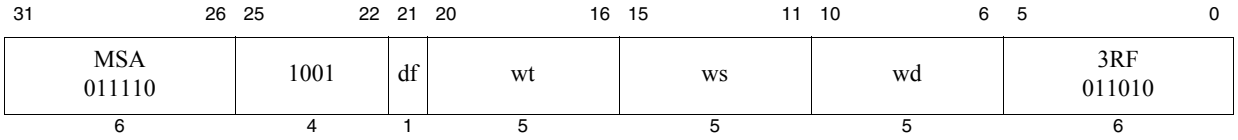
Data-dependent exceptions are possible as specified by the IEEE Standard for Floating-Point Arithmetic 754™-2008.

**Operation:**

```
FSULT.W
    for i in 0 .. WRLEN/32-1
        c ← UnorderedSigFP(WR[ws]32i+31..32i, WR[wt]32i+31..32i, 32)
        d ← LessSigFP(WR[ws]32i+31..32i, WR[wt]32i+31..32i, 32)
        WR[wd]32i+31..32i ← (c | d)32
    endfor

FSULT.D
    for i in 0 .. WRLEN/64-1
        c ← UnorderedSigFP(WR[ws]64i+63..64i, WR[wt]64i+63..64i, 64)
        d ← LessSigFP(WR[ws]64i+63..64i, WR[wt]64i+63..64i, 64)
        WR[wd]64i+63..64i ← (c | d)64
    endfor

function UnorderedSigFP(tt, ts, n)
    /* Implementation defined signaling unordered compare operation. */
endfunction UnorderedSigFP

function LessThanSigFP(tt, ts, n)
    /* Implementation defined signaling less than compare operation. */
endfunction LessThanSigFP
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception, MSA Floating Point Exception.




**Format:** `FSUN.df`

`FSUN.W wd,ws,wt` **MSA**

`FSUN.D wd,ws,wt` **MSA**

**Purpose:** Vector Floating-Point Signaling Compare Unordered

Vector to vector floating-point signaling compare unordered; if true all destination bits are set, otherwise clear.

**Description:** `wd[i] ← (ws[i] ?(signaling) wt[i])`

Set all bits to 1 in *wd* elements if the corresponding *ws* and *wt* floating-point elements are unordered, i.e. at least one element is a NaN value, otherwise set all bits to 0.

The signaling compare operation is defined by the IEEE Standard for Floating-Point Arithmetic 754™-2008.

The Inexact Exception is not signaled when subnormal input operands are flushed based on the flush-to-zero bit FS in MSA Control and Status Register *MSACSR*. In case of a floating-point exception, the default result has all bits set to 1.

The operands are values in floating-point data format *df*. The results are values in integer data format *df*.

**Restrictions:**

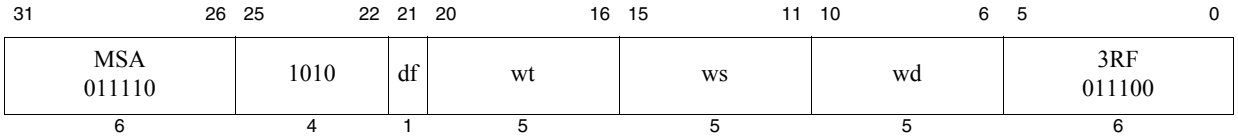
Data-dependent exceptions are possible as specified by the IEEE Standard for Floating-Point Arithmetic 754™-2008.

**Operation:**

```
FSUN.W
    for i in 0 .. WRLEN/32-1
        c ← UnorderedSigFP(WR[ws]32i+31..32i, WR[wt]32i+31..32i, 32)
        WR[wd]32i+31..32i ← c32
    endfor

FSUN.D
    for i in 0 .. WRLEN/64-1
        c ← UnorderedSigFP(WR[ws]64i+63..64i, WR[wt]64i+63..64i, 64)
        WR[wd]64i+63..64i ← c64
    endfor

function UnorderedSigFP(tt, ts, n)
    /* Implementation defined signaling unordered compare operation. */
endfunction UnorderedSigFP
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception, MSA Floating Point Exception.




**Format:** `FSUNE.df`

`FSUNE.W wd,ws,wt` **MSA**

`FSUNE.D wd,ws,wt` **MSA**

**Purpose:** Vector Floating-Point Signaling Compare Unordered or Not Equal

Vector to vector floating-point signaling compare for unordered or not equal; if true all destination bits are set, otherwise clear.

**Description:** `wd[i] ← (ws[i] ≠?(signaling) wt[i])`

Set all bits to 1 in *wd* elements if the corresponding *ws* and *wt* floating-point elements are unordered or not equal, otherwise set all bits to 0.

The signaling compare operation is defined by the IEEE Standard for Floating-Point Arithmetic 754™-2008.

The Inexact Exception is not signaled when subnormal input operands are flushed based on the flush-to-zero bit FS in MSA Control and Status Register *MSACSR*. In case of a floating-point exception, the default result has all bits set to 1.

The operands are values in floating-point data format *df*. The results are values in integer data format *df*.

**Restrictions:**

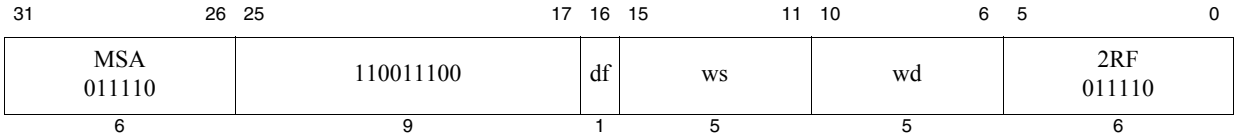
Data-dependent exceptions are possible as specified by the IEEE Standard for Floating-Point Arithmetic 754™-2008.

**Operation:**

```
FSUNE.W
    for i in 0 .. WRLEN/32-1
        c ← UnorderedSigFP(WR[ws]32i+31..32i, WR[wt]32i+31..32i, 32)
        d ← NotEqualSigFP(WR[ws]32i+31..32i, WR[wt]32i+31..32i, 32)
        WR[wd]32i+31..32i ← (c | d)32
    endfor

FSUNE.D
    for i in 0 .. WRLEN/64-1
        c ← UnorderedSigFP(WR[ws]64i+63..64i, WR[wt]64i+63..64i, 64)
        c ← NotEqualSigFP(WR[ws]64i+63..64i, WR[wt]64i+63..64i, 64)
        WR[wd]64i+63..64i ← (c | d)64
    endfor

function UnorderedSigFP(tt, ts, n)
    /* Implementation defined signaling unordered compare operation. */
endfunction UnorderedSigFP

function NotEqualSigFP(tt, ts, n)
    /* Implementation defined signaling not equal compare operation. */
endfunction NotEqualSigFP
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception, MSA Floating Point Exception.




**Format:** `FTINT_S.df`

`FTINT_S.W wd,ws` **MSA**

`FTINT_S.D wd,ws` **MSA**

**Purpose:** Vector Floating-Point Convert to Signed Integer

Vector floating-point convert to signed integer.

**Description:** `wd[i] ← to_int_s(ws[i])`

The floating-point elements in *ws* are rounded and converted to signed integer values based on the rounding mode bits RM in MSA Control and Status Register *MSACSR*. The result is written to vector *wd*.

The floating-point to integer conversion operation is exact as defined by the IEEE Standard for Floating-Point Arithmetic 754™-2008, i.e. the Inexact exception is signaled if the result does not have the same numerical value as the input operand. In this case, the default result is the rounded result.

NaN values and numeric operands converting to an integer outside the range of the destination format signal the Invalid Operation exception. For positive numeric operands outside the range, the default result is the largest signed integer value. The default result for negative numeric operands outside the range is the smallest signed integer value. The default result for NaN operands is zero.

The operands are values in floating-point data format *df*. The results are values in integer data format *df*.

**Restrictions:**

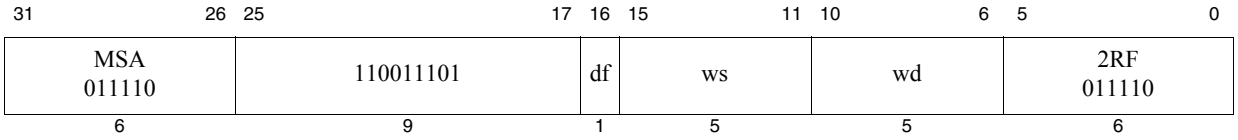
Data-dependent exceptions are possible.

**Operation:**

```
FTINT_S.W
   for i in 0 .. WRLEN/32-1
      f ← ToIntSignedFP(WR[ws]32i+31..32i, 32)
      WR[wd]32i+31..32i ← f
   endfor

FTINT_S.D
   for i in 0 .. WRLEN/64-1
      f ← ToIntSignedFP(WR[ws]64i+63..64i, 64)
      WR[wd]64i+63..64i ← f
   endfor

function ToIntSignedFP(tt, n)
   /* Implementation defined floating-point rounding and signed
         integer conversion. */
endfunction ToIntSignedFP
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception, MSA Floating Point Exception.




**Format:** `FTINT_U.df`

`FTINT_U.W wd,ws` **MSA**

`FTINT_U.D wd,ws` **MSA**

**Purpose:** Vector Floating-Point Round and Convert to Unsigned Integer

Vector floating-point round and convert to unsigned integer.

**Description:** `wd[i] ← to_int_u(ws[i])`

The floating-point elements in *ws* are rounded and converted to unsigned integer values based on the rounding mode bits RM in MSA Control and Status Register *MSACSR*. The result is written to vector *wd*.

The floating-point to integer conversion operation is exact as defined by the IEEE Standard for Floating-Point Arithmetic 754™-2008, i.e. the Inexact exception is signaled if the result does not have the same numerical value as the input operand. In this case, the default result is the rounded result.

NaN values and numeric operands converting to an integer outside the range of the destination format signal the Invalid Operation exception. For positive numeric operands outside the range, the default result is the largest unsigned integer value. The default result for negative numeric operands is zero. The default result for NaN operands is zero.

The operands are values in floating_point data format *df*. The results are values in integer data format *df*.

**Restrictions:**

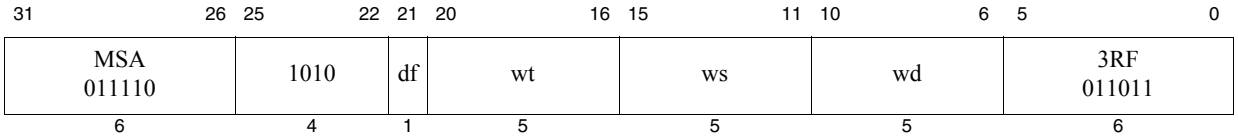
Data-dependent exceptions are possible.

**Operation:**

```
FTINT_U.W
    for i in 0 .. WRLEN/32-1
        f ← ToIntUnsignedFP(WR[ws]32i+31..32i, 32)
        WR[wd]32i+31..32i ← f
    endfor

FTINT_U.D
    for i in 0 .. WRLEN/64-1
        f ← ToIntUnsignedFP(WR[ws]64i+63..64i, 64)
        WR[wd]64i+63..64i ← f
    endfor

function ToIntUnsignedFP(tt, n)
    /* Implementation defined floating-point rounding and unsigned
           integer conversion. */
endfunction ToIntUnsignedFP
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception, MSA Floating Point Exception.




**Format:** `FTQ.df`

`FTQ.H wd,ws,wt` **MSA**

`FTQ.W wd,ws,wt` **MSA**

**Purpose:** Vector Floating-Point Convert to Fixed-Point

Vector fix-point format conversion from floating-point.

**Description:** `left_half(wd)[i] ← to_q(ws[i]); right_half(wd)[i] ← to_q(wt[i])`

The floating-point elements in vectors *ws* and *wt* are down-converted to a fixed-point representation, i.e. from 64-bit floating-point to 32-bit Q31 fixed-point representation, or from 32-bit floating-point to 16-bit Q15 fixed-point representation.

The floating-point data inside the fixed-point range is first scaled up (multiplied by $2^{15}$ or $2^{31}$) and then rounded and converted to a 16-bit or 32 -bit integer based on the rounding mode bits RM in MSA Control and Status Register *MSACSR*. The resulting value is the Q15 or Q31 representation.

The scaling and floating-point to integer conversion operations are defined by the IEEE Standard for Floating-Point Arithmetic 754™-2008. The integer conversion operation is exact, i.e. the Inexact exception is signaled if the result does not have the same numerical value as the input operand. In this case, the default result is the rounded result.

NaN values signal the Invalid Operation exception. Numeric operands converting to fixed-point values outside the range of the destination format signal the Overflow and the Inexact exceptions. For positive numeric operands outside the range, the default result is the largest fixed-point value. The default result for negative numeric operands outside the range is the smallest fixed-point value. The default result for NaN operands is zero.

The operands are values in floating-point data format *df*. The results are fixed-point values in data format half the size of *df*.

**Restrictions:**

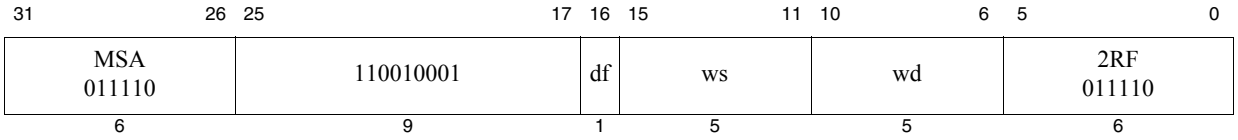
Data-dependent exceptions are possible.

**Operation:**

```
FTQ.H
    for i in 0 .. WRLEN/32-1
        q ← ToFixPointFP((WR[ws]32i+31..32i, 32)
        r ← ToFixPointFP((WR[wt]32i+31..32i, 32)
        WR[wd]16i+15+WRLEN/2..16i+WRLEN/2 ← q
        WR[wd]16i+15..16i ← r
    endfor

FTQ.W
    for i in 0 .. WRLEN/64-1
        q ← ToFixPointFP((WR[ws]64i+63..64i, 64)
        r ← ToFixPointFP((WR[wt]64i+63..64i, 64)
        WR[wd]32i+31+WRLEN/2..32i+WRLEN/2 ← q
        WR[wd]32i+31..32i ← r
    endfor
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception, MSA Floating Point Exception.




**Format:** `FTRUNC_S.df`

`FTRUNC_S.W wd,ws` **MSA**

`FTRUNC_S.D wd,ws` **MSA**

**Purpose:** Vector Floating-Point Truncate and Convert to Signed Integer

Vector floating-point truncate and convert to signed integer.

**Description:** `wd[i] ← truncate_to_int_s(ws[i])`

The floating-point elements in *ws* are truncated, i.e. rounded toward zero, to signed integer values. The rounding mode bits RM in MSA Control and Status Register *MSACSR* are not used. The result is written to vector *wd*.

The floating-point to integer conversion operation is exact as defined by the IEEE Standard for Floating-Point Arithmetic 754™-2008, i.e. the Inexact exception is signaled if the result does not have the same numerical value as the input operand. In this case, the default result is the rounded result.

NaN values and numeric operands converting to an integer outside the range of the destination format signal the Invalid Operation exception. For positive numeric operands outside the range, the default result is the largest signed integer value. The default result for negative numeric operands outside the range is the smallest signed integer value. The default result for NaN operands is zero.

The operands are values in floating-point data format *df*. The results are values in integer data format *df*.

**Restrictions:**

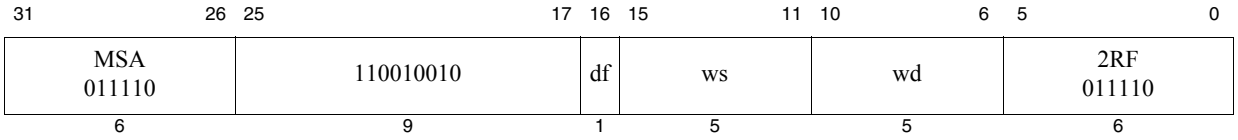
Data-dependent exceptions are possible.

**Operation:**

```
FTRUNC_S.W
    for i in 0 .. WRLEN/32-1
        f ← TruncToIntSignedFP(WR[ws]32i+31..32i, 32)
        WR[wd]32i+31..32i ← f
    endfor

FTRUNC_S.D
    for i in 0 .. WRLEN/64-1
        f ← TruncToIntSignedFP(WR[ws]64i+63..64i, 64)
        WR[wd]64i+63..64i ← f
    endfor

function TruncToIntSignedFP(tt, n)
    /* Implementation defined floating-point truncation and signed
           integer conversion. */
endfunction TruncToIntSignedFP
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception, MSA Floating Point Exception.




**Format:** `FTRUNC_U.df`

`FTRUNC_U.W wd,ws` **MSA**

`FTRUNC_U.D wd,ws` **MSA**

**Purpose:** Vector Floating-Point Truncate and Convert to Unsigned Integer

Vector floating-point truncate and convert to unsigned integer.

**Description:** `wd[i] ← truncate_to_int_u(ws[i])`

The floating-point elements in *ws* are truncated, i.e. rounded toward zero, to unsigned integer values. The rounding mode bits RM in MSA Control and Status Register *MSACSR* are not used. The result is written to vector *wd*.

The floating-point to integer conversion operation is exact as defined by the IEEE Standard for Floating-Point Arithmetic 754™-2008, i.e. the Inexact exception is signaled if the result does not have the same numerical value as the input operand. In this case, the default result is the rounded result.

NaN values and numeric operands converting to an integer outside the range of the destination format signal the Invalid Operation exception. For positive numeric operands outside the range, the default result is the largest unsigned integer value. The default value for negative numeric operands is zero. The default result for NaN operands is zero.

The operands are values in floating_point data format *df*. The results are values in integer data format *df*.

**Restrictions:**

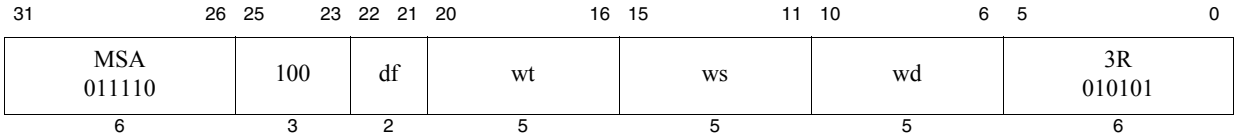
Data-dependent exceptions are possible.

**Operation:**

```
FTRUNC_U.W
   for i in 0 .. WRLEN/32-1
      f ← TruncToIntUnsignedFP(WR[ws]32i+31..32i, 32)
      WR[wd]32i+31..32i ← f
   endfor

FTRUNC_U.D
   for i in 0 .. WRLEN/64-1
      f ← TruncToIntUnsignedFP(WR[ws]64i+63..64i, 64)
      WR[wd]64i+63..64i ← f
   endfor

function TruncToIntUnsignedFP(tt, n)
   /* Implementation defined floating-point truncation and unsigned
         integer conversion. */
endfunction TruncToIntUnsignedFP
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception, MSA Floating Point Exception.




**Format:** `HADD_S.df`

`HADD_S.H wd,ws,wt` **MSA**

`HADD_S.W wd,ws,wt` **MSA**

`HADD_S.D wd,ws,wt` **MSA**

**Purpose:** Vector Signed Horizontal Add

Vector sign extend and pairwise add the odd elements with the even elements to double width elements

**Description:** `(wd[2i+1], wd[2i]) ← signed(ws[2i+1]) + signed(wt[2i])`

The sign-extended odd elements in vector *ws* are added to the sign-extended even elements in vector *wt* producing a result twice the size of the input operands. The result is written to vector *wd*.

The operands are values in integer data format half the size of *df*. The results are values in integer data format *df*.

**Restrictions:**

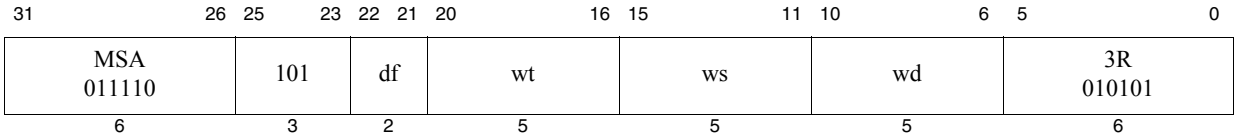
No data-dependent exceptions are possible.

**Operation:**

```
HADD_S.H
    for i in 0 .. WRLEN/16-1
        WR[wd]16i+15..16i ← hadd_s(WR[ws]16i+15..16i, WR[wt]16i+15..16i, 8)
    endfor

HADD_S.W
    for i in 0 .. WRLEN/32-1
        WR[wd]32i+31..32i ← hadd_s(WR[ws]32i+31..32i, WR[wt]32i+31..32i, 16)
    endfor

HADD_S.D
    for i in 0 .. WRLEN/64-1
        WR[wd]64i+63..64i ← hadd_s(WR[ws]64i+63..64i, WR[wt]64i+63..64i, 32)
    endfor

function hadd_s(ts, tt, n)
    t ← ((ts2n-1)^n || ts2n-1..n) + ((ttn-1)^n || ttn-1..0)
    return t
endfunction hadd_s
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception.




**Format:** `HADD_U.df`

`HADD_U.H wd,ws,wt` **MSA**

`HADD_U.W wd,ws,wt` **MSA**

`HADD_U.D wd,ws,wt` **MSA**

**Purpose:** Vector Unsigned Horizontal Add

Vector zero extend and pairwise add the odd elements with the even elements to double width elements

**Description:** `(wd[2i+1], wd[2i]) ← unsigned(ws[2i+1]) + unsigned(wt[2i])`

The zero-extended odd elements in vector *ws* are added to the zero-extended even elements in vector *wt* producing a result twice the size of the input operands. The result is written to vector *wd*.

The operands are values in integer data format half the size of *df*. The results are values in integer data format *df*.

**Restrictions:**

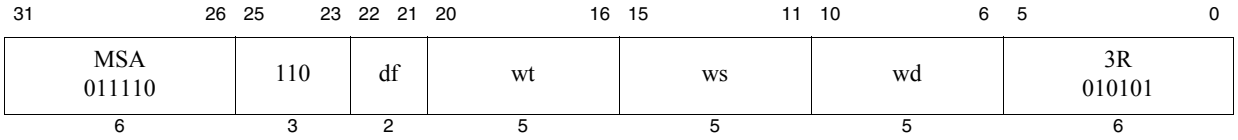
No data-dependent exceptions are possible.

**Operation:**

```
HADD_U.H
    for i in 0 .. WRLEN/16-1
        WR[wd]16i+15..16i ← hadd_u(WR[ws]16i+15..16i, WR[wt]16i+15..16i, 8)
    endfor

HADD_U.W
    for i in 0 .. WRLEN/32-1
        WR[wd]32i+31..32i ← hadd_u(WR[ws]32i+31..32i, WR[wt]32i+31..32i, 16)
    endfor

HADD_U.D
    for i in 0 .. WRLEN/64-1
        WR[wd]64i+63..64i ← hadd_u(WR[ws]64i+63..64i, WR[wt]64i+63..64i, 32)
    endfor

function hadd_u(ts, tt, n)
    t ← (0^n || ts2n-1..n) + (0^n || ttn-1..0)
    return t
endfunction hadd_u
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception.




**Format:** `HSUB_S.df`

`HSUB_S.H wd,ws,wt` **MSA**

`HSUB_S.W wd,ws,wt` **MSA**

`HSUB_S.D wd,ws,wt` **MSA**

**Purpose:** Vector Signed Horizontal Subtract

Vector sign extend and pairwise subtract the even elements from the odd elements to double width elements

**Description:** `(wd[2i+1], wd[2i]) ← signed(ws[2i+1]) - signed(wt[2i])`

The sign-extended odd elements in vector *wt* are subtracted from the sign-extended even elements in vector *wt* producing a signed result twice the size of the input operands. The result is written to vector *wd*.

The operands are values in integer data format half the size of *df*. The results are values in integer data format *df*.

**Restrictions:**

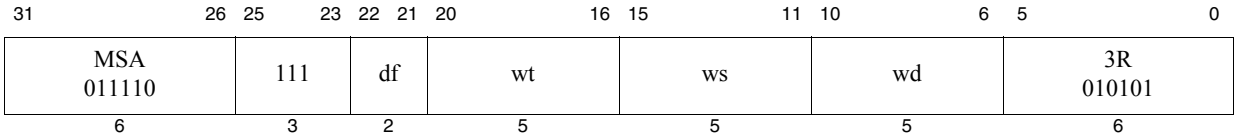
No data-dependent exceptions are possible.

**Operation:**

```
HSUB_S.H
    for i in 0 .. WRLEN/16-1
        WR[wd]16i+15..16i ← hsub_s(WR[ws]16i+15..16i, WR[wt]16i+15..16i, 8)
    endfor

HSUB_S.W
    for i in 0 .. WRLEN/32-1
        WR[wd]32i+31..32i ← hsub_s(WR[ws]32i+31..32i, WR[wt]32i+31..32i, 16)
    endfor

HSUB_S.D
    for i in 0 .. WRLEN/64-1
        WR[wd]64i+63..64i ← hsub_s(WR[ws]64i+63..64i, WR[wt]64i+63..64i, 32)
    endfor

function hsub_s(ts, tt, n)
    t ← ((ts2n-1)^n || ts2n-1..n) - ((ttn-1)^n || ttn-1..0)
    return t
endfunction hsub_s
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception.




**Format:** `HSUB_U.df`

`HSUB_U.H wd,ws,wt` **MSA**

`HSUB_U.W wd,ws,wt` **MSA**

`HSUB_U.D wd,ws,wt` **MSA**

**Purpose:** Vector Unsigned Horizontal Subtract

Vector zero extend and pairwise subtract the even elements from the odd elements to double width elements

**Description:** `(wd[2i+1], wd[2i]) ← unsigned(ws[2i+1]) - unsigned(wt[2i])`

The zero-extended odd elements in vector *wt* are subtracted from the zero-extended even elements in vector *ws* producing a signed result twice the size of the input operands. The result is written to vector *wd*.

The operands are values in integer data format half the size of *df*. The results are values in integer data format *df*.

**Restrictions:**

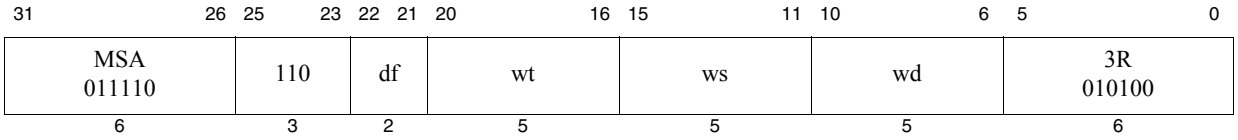
No data-dependent exceptions are possible.

**Operation:**

```
HSUB_U.H
    for i in 0 .. WRLEN/16-1
        WR[wd]16i+15..16i ← hsub_u(WR[ws]16i+15..16i, WR[wt]16i+15..16i, 8)
    endfor

HSUB_U.W
    for i in 0 .. WRLEN/32-1
        WR[wd]32i+31..32i ← hsub_u(WR[ws]32i+31..32i, WR[wt]32i+31..32i, 16)
    endfor

HSUB_U.D
    for i in 0 .. WRLEN/64-1
        WR[wd]64i+63..64i ← hsub_u(WR[ws]64i+63..64i, WR[wt]64i+63..64i, 32)
    endfor

function hsub_u(ts, tt, n)
    t ← (0^n || ts2n-1..n) - (0^n || ttn-1..0)
    return t
endfunction hsub_u
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception.




**Format:** `ILVEV.df`

`ILVEV.B wd,ws,wt` **MSA**

`ILVEV.H wd,ws,wt` **MSA**

`ILVEV.W wd,ws,wt` **MSA**

`ILVEV.D wd,ws,wt` **MSA**

**Purpose:** Vector Interleave Even

Vector even elements interleave.

**Description:** `wd[2i] ← wt[2i]; wd[2i+1] ← ws[2i]`

Even elements in v ectors *ws* and *wt* are copied to vector *wd* alternating one element from *ws* with one element from *wt*.

The operands and results are values in integer data format *df*.

**Restrictions:**

No data-dependent exceptions are possible.

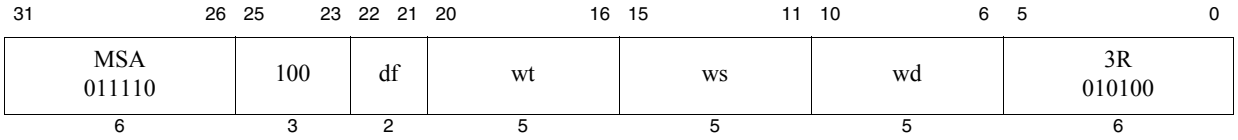
**Operation:**

```
ILVEV.B
   for i in 0 .. WRLEN/16-1
      j ← 2 * i
      k ← 2 * i + 1
      WR[wd]8j+7..8j ← WR[wt]8j+7..8j
      WR[wd]8k+7..8k ← WR[ws]8j+7..8j
   endfor

ILVEV.H
   for i in 0 .. WRLEN/32-1
      j ← 2 * i
      k ← 2 * i + 1
      WR[wd]16j+15..16j ← WR[wt]16j+15..16j
      WR[wd]16k+15..16k ← WR[ws]16j+15..16j
   endfor

ILVEV.W
   for i in 0 .. WRLEN/64-1
      j ← 2 * i
      k ← 2 * i + 1
      WR[wd]32j+31..32j ← WR[wt]32j+31..32j
      WR[wd]32k+31..32k ← WR[ws]32j+31..32j
   endfor

ILVEV.D
   for i in 0 .. WRLEN/128-1
      j ← 2 * i
      k ← 2 * i + 1
      WR[wd]64j+63..64j ← WR[wt]64j+63..64j
      WR[wd]64k+63..64k ← WR[ws]64j+63..64j
```

```
    endfor
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception.




**Format:** `ILVL.df`

`ILVL.B wd,ws,wt` **MSA**

`ILVL.H wd,ws,wt` **MSA**

`ILVL.W wd,ws,wt` **MSA**

`ILVL.D wd,ws,wt` **MSA**

**Purpose:** Vector Interleave Left

Vector left elements interleave.

**Description:** `wd[2i] ← left_half(wt)[i]; wd[2i+1] ← left_half(ws)[i]`

The left half elements in vectors *ws* and *wt* are copied to vector *wd* alternating one element from *ws* with one element from *wt*.

The operands and results are values in integer data format *df*.

**Restrictions:**

No data-dependent exceptions are possible.

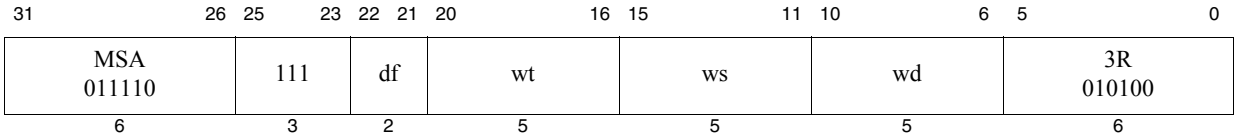
**Operation:**

```
ILVL.B
   for i in 0 .. WRLEN/16-1
      j ← 2 * i
      k ← 2 * i + 1
      WR[wd]8j+7..8j ← WR[wt]8i+7+WRLEN/2..8i+WRLEN/2
      WR[wd]8k+7..8k ← WR[ws]8i+7+WRLEN/2..8i+WRLEN/2
   endfor

ILVL.H
   for i in 0 .. WRLEN/32-1
      j ← 2 * i
      k ← 2 * i + 1
      WR[wd]16j+15..16j ← WR[wt]16i+15+WRLEN/2..16i+WRLEN/2
      WR[wd]16k+15..16k ← WR[ws]16i+15+WRLEN/2..16i+WRLEN/2
   endfor

ILVL.W
   for i in 0 .. WRLEN/64-1
      j ← 2 * i
      k ← 2 * i + 1
      WR[wd]32j+31..32j ← WR[wt]32i+31+WRLEN/2..32i+WRLEN/2
      WR[wd]32k+31..32k ← WR[ws]32i+31+WRLEN/2..32i+WRLEN/2
   endfor

ILVL.D
   for i in 0 .. WRLEN/128-1
      j ← 2 * i
      k ← 2 * i + 1
      WR[wd]64j+63..64j ← WR[wt]64i+63+WRLEN/2..64i+WRLEN/2
      WR[wd]64k+63..64k ← WR[ws]64i+63+WRLEN/2..64i+WRLEN/2
```

```
    endfor
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception.




**Format:** `ILVOD.df`

`ILVOD.B wd,ws,wt` **MSA**

`ILVOD.H wd,ws,wt` **MSA**

`ILVOD.W wd,ws,wt` **MSA**

`ILVOD.D wd,ws,wt` **MSA**

**Purpose:** Vector Interleave Odd

Vector odd elements interleave.

**Description:** `wd[2i] ← wt[2i+1]; wd[2i+1] ← ws[2i+1]`

Odd elements in v ectors *ws* and *wt* are copied to v ector *wd* alternating one element from *ws* with one element from *wt*.

The operands and results are values in integer data format *df*.

**Restrictions:**

No data-dependent exceptions are possible.

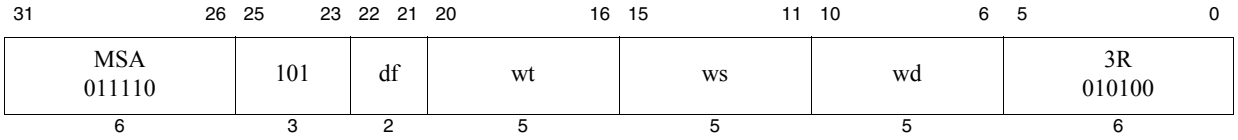
**Operation:**

```
ILVOD.B
   for i in 0 .. WRLEN/16-1
      j ← 2 * i
      k ← 2 * i + 1
      WR[wd]8j+7..8j ← WR[wt]8k+7..8k
      WR[wd]8k+7..8k ← WR[ws]8k+7..8k
   endfor

ILVOD.H
   for i in 0 .. WRLEN/32-1
      j ← 2 * i
      k ← 2 * i + 1
      WR[wd]16j+15..16j ← WR[wt]16k+15..16k
      WR[wd]16k+15..16k ← WR[ws]16k+15..16k
   endfor

ILVOD.W
   for i in 0 .. WRLEN/64-1
      j ← 2 * i
      k ← 2 * i + 1
      WR[wd]32j+31..32j ← WR[wt]32k+31..32k
      WR[wd]32k+31..32k ← WR[ws]32k+31..32k
   endfor

ILVOD.D
   for i in 0 .. WRLEN/128-1
      j ← 2 * i
      k ← 2 * i + 1
      WR[wd]64j+63..64j ← WR[wt]64k+63..64k
      WR[wd]64k+63..64k ← WR[ws]64k+63..64k
```

```
    endfor
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception.




**Format:** `ILVR.df`

`ILVR.B wd,ws,wt` **MSA**

`ILVR.H wd,ws,wt` **MSA**

`ILVR.W wd,ws,wt` **MSA**

`ILVR.D wd,ws,wt` **MSA**

**Purpose:** Vector Interleave Right

Vector right elements interleave.

**Description:** `wd[2i] ← right_half(wt)[i]; wd[2i+1] ← right_half(ws)[i]`

The right half elements in vectors *ws* and *wt* are copied to vector *wd* alternating one element from *ws* with one element from *wt*.

The operands and results are values in integer data format *df*.

**Restrictions:**

No data-dependent exceptions are possible.

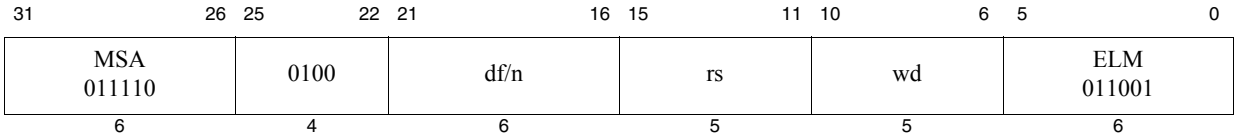
**Operation:**

```
ILVR.B
    for i in 0 .. WRLEN/16-1
        j ← 2 * i
        k ← 2 * i + 1
        WR[wd]8j+7..8j ← WR[wt]8i+7..8i
        WR[wd]8k+7..8k ← WR[ws]8i+7..8i
    endfor

ILVR.H
    for i in 0 .. WRLEN/32-1
        j ← 2 * i
        k ← 2 * i + 1
        WR[wd]16j+15..16j ← WR[wt]16i+15..16i
        WR[wd]16k+15..16k ← WR[ws]16i+15..16i
    endfor

ILVR.W
    for i in 0 .. WRLEN/64-1
        j ← 2 * i
        k ← 2 * i + 1
        WR[wd]32j+31..32j ← WR[wt]32i+31..32i
        WR[wd]32k+31..32k ← WR[ws]32i+31..32i
    endfor

ILVR.D
    for i in 0 .. WRLEN/128-1
        j ← 2 * i
        k ← 2 * i + 1
        WR[wd]64j+63..64j ← WR[wt]64i+63..64i
        WR[wd]64k+63..64k ← WR[ws]64i+63..64i
```

```
    endfor
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception.




**Format:** `INSERT.df`

`INSERT.B wd[n],rs` **MSA**

`INSERT.H wd[n],rs` **MSA**

`INSERT.W wd[n],rs` **MSA**

`INSERT.D wd[n],rs` **MIPS64 MSA**

**Purpose:** GPR Insert Element

GPR value copied to vector element.

**Description:** `wd[n] ← rs`

Set element *n* in vector *wd* to GPR *rs* value. All other elements in vector *wd* are unchanged. If the source GPR is wider than the destination data format, the destination's elements will be set to the least significant bits of the GPR.

The operands and results are values in data format *df*.

**Restrictions:**

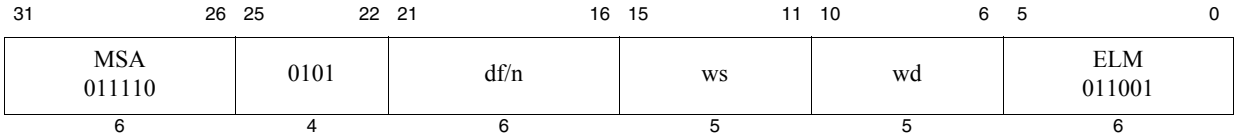
No data-dependent exceptions are possible.

**Operation:**

```
INSERT.B
    WR[wd]8n+7..8n ← GPR[rs]7..0

INSERT.H
    WR[wd]16n+15..16n ← GPR[rs]15..0

INSERT.W
    WR[wd]32n+31..32n ← GPR[rs]31..0

INSERT.D
    WR[wd]64n+63..64n ← GPR[rs]63..0
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception.




**Format:** `INSVE.df`

`INSVE.B wd[n],ws[0]` **MSA**

`INSVE.H wd[n],ws[0]` **MSA**

`INSVE.W wd[n],ws[0]` **MSA**

`INSVE.D wd[n],ws[0]` **MSA**

**Purpose:** Element Insert Element

Element value copied to vector element.

**Description:** `wd[n] ← ws[0]`

Set element *n* in vector *wd* to element 0 in vector *ws* value. All other elements in vector *wd* are unchanged.

The operands and results are values in data format *df*.

**Restrictions:**

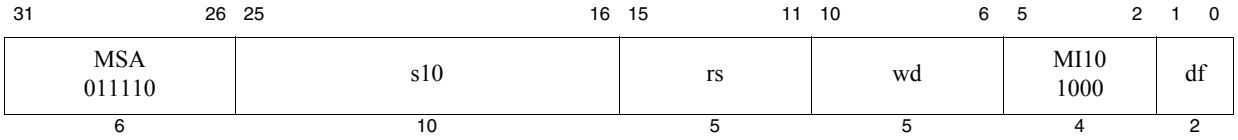
No data-dependent exceptions are possible.

**Operation:**

```
INSVE.B
    WR[wd]8n+7..8n ← WR[ws]7..0

INSVE.H
    WR[wd]16n+15..16n ← WR[ws]15..0

INSVE.W
    WR[wd]32n+31..32n ← WR[ws]31..0

INSVE.D
    WR[wd]64n+63..64n ← WR[ws]63..0
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception.




**Format:** `LD.df`

`LD.B wd,s10(rs)` **MSA**

`LD.H wd,s10(rs)` **MSA**

`LD.W wd,s10(rs)` **MSA**

`LD.D wd,s10(rs)` **MSA**

**Purpose:** Vector Load

Vector load element-by-element from base register plus offset memory address,

**Description:** `wd[i] ← memory[rs + (s10 + i) * sizeof(wd[i])]`

The *WRLEN* / 8 bytes at the ef fective memory location addressed by the base *rs* and the 10-bit signed immediate offset *s10* are fetched and placed in *wd* as elements of data format *df*.

The *s10* offset in data format *df* units is added to the base *rs* to form the effective memory location address. *rs* and the effective memory location address have no alignment restrictions.

If the effective memory location address is element aligned, the vector load instruction is atomic at the element level with no guaranteed ordering among elemen ts, i.e. each element load is an ato mic operation issued in no particular order with respect to the element's vector position.

By convention, in the assembly language syntax all offsets are in bytes and have to be multiple of the size of the data format *df*. The assembler determines the *s10* bitfield value dividing the byte offset by the size of the data format *df*.

**Restrictions:**

Address-dependent exceptions are possible.

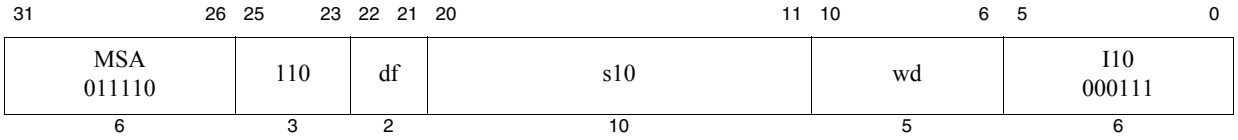
**Operation:**

```
LD.B
   a ← rs + s10
   LoadByteVector(WR[wd]WRLEN-1..0, a, WRLEN/8)

LD.H
   a ← rs + s10 * 2
   LoadHalfwordVector(WR[wd]WRLEN-1..0, a, WRLEN/16)

LD.W
   a ← rs + s10 * 4
   LoadWordVector(WR[wd]WRLEN-1..0, a, WRLEN/32)

LD.D
   a ← rs + s10 * 8
   LoadDoublewordVector(WR[wd]WRLEN-1..0, a, WRLEN/64)

function LoadByteVector(ts, a, n)
   /* Implementation defined load ts vector of n bytes from virtual
         address a. */
endfunction LoadByteVector

function LoadHalfwordVector(ts, a, n)
   /* Implementation defined load ts vector of n halfwords from
```

```
          virtual address a. */
endfunction LoadHalfwordVector

function LoadWordVector(ts, a, n)
   /* Implementation defined load ts vector of n words from virtual
          address a. */
endfunction LoadWordVector

function LoadDoublewordVector(ts, a, n)
   /* Implementation defined load ts vector of n doublewords from
          virtual address a. */
endfunction LoadDoublewordVector
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception. Data access TLB and Address Error Exceptions.




**Format:** `LDI.df`

`LDI.B wd,s10` **MSA**

`LDI.H wd,s10` **MSA**

`LDI.W wd,s10` **MSA**

`LDI.D wd,s10` **MSA**

**Purpose:** Immediate Load

Immediate value replicated across all destination elements.

**Description:** `wd[i] ← s10`

The signed immediate s10 is replicated in all *wd* elements. For byte elements, only the least significant 8 bits of s10 will be used.

**Restrictions:**

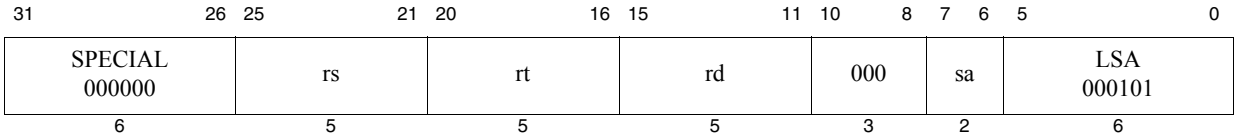
No data-dependent exceptions are possible.

**Operation:**

```
LDI.B
    t ← s10_7..0
    for i in 0 .. WRLEN/8-1
        WR[wd]_8i+7..8i ← t
    endfor

LDI.H
    t ← (s10_9)^6 || s10_9..0
    for i in 0 .. WRLEN/16-1
        WR[wd]_16i+15..16i ← t
    endfor

LDI.W
    t ← (s10_9)^22 || s10_9..0
    for i in 0 .. WRLEN/32-1
        WR[wd]_32i+31..32i ← t
    endfor

LDI.D
    t ← (s10_9)^54 || s10_9..0
    for i in 0 .. WRLEN/64-1
        WR[wd]_64i+63..64i ← t
    endfor
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception.




**Format:** `LSA`

`LSA rd,rs,rt,sa` **MSA**

**Purpose:** Left Shift Add

To left-shift a word by a fixed number of bits and add the result to another word.

**Description:** `GPR[rd] ← (GPR[rs] << (sa + 1)) + GPR[rt]`

The 32-bit word value in GPR *rs* is shifted left, inserting zeros into the emptied bits; the 32-bit word result is added to the 32-bit value in GPR *rt* and the 32-bit arithmetic result is sign-extended and placed into GPR *rd*.

No Integer Overflow exception occurs under any circumstances.

**Restrictions:**

A Reserved Instruction Exception is signaled if MSA implementation is not present.

If GPR *rt* does not contain sign-extended 32-bit values (bits 63..31 equal), then the result of the operation is **UNPREDICTABLE**.

**Operation:**

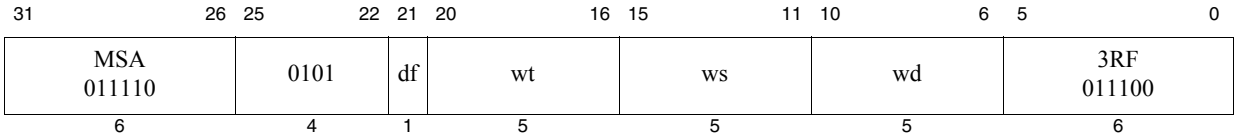
```
if NotWordValue(GPR[rt]) then
   UNPREDICTABLE
endif
if Config3MSAP = 1 then
   s ← sa + 1
   temp ← (GPR[rs](31-s)..0 || 0s) + GPR[rt]
   GPR[rd] ← sign_extend(temp31..0)
else
   SignalException(ReservedInstruction)
endif
```

**Exceptions:**

Reserved Instruction Exception.

**Programming Notes:**

Unlike nearly all other word operations, the LSA input operand GPR *rs* does not have to be a properly sign-extended word value to produce a valid sign-extended 32-bit result. The result word is always sign-extended into a 64-bit destination register.




**Format:** `MADD_Q.df`

`MADD_Q.H wd,ws,wt` **MSA**

`MADD_Q.W wd,ws,wt` **MSA**

**Purpose:** Vector Fixed-Point Multiply and Add

Vector fixed-point multiply and add.

**Description:** `wd[i] ← saturate(wd[i] + ws[i] * wt[i])`

The products of fixed-point elements in vector *wt* by fixed-point elements in vector *ws* are added to the fixed-point elements in vector *wd*. The multiplication result is not saturated, i.e. exact (-1) * (-1) = 1 is added to the destination. The saturated fixed-point results are stored back to *wd*.

Internally, the multiplication and addition operate on data double the size of *df*. Truncation to fixed-point data format *df* is performed at the very last stage, after saturation.

The operands and results are values in fixed-point data format *df*.

**Restrictions:**

No data-dependent exceptions are possible.

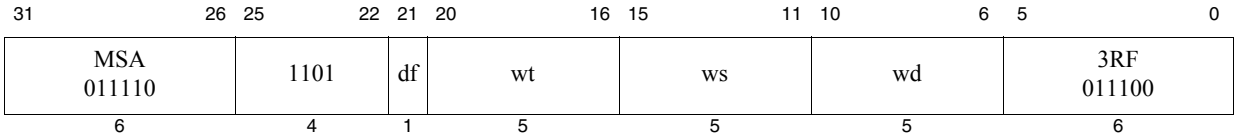
**Operation:**

```
MADD_Q.H
    for i in 0 .. WRLEN/16-1
        WR[wd]16i+15..16i ←
                q_madd(WR[wd]16i+15..16i, WR[ws]16i+15..16i, WR[wt]16i+15..16i, 16)
    endfor

MADD_Q.W
    for i in 0 .. WRLEN/32-1
        WR[wd]32i+31..32i ←
                q_madd(WR[wd]32i+31..32i, WR[ws]32i+31..32i, WR[wt]32i+31..32i, 32)
    endfor

function mulx_s(ts, tt, n)
    s ← (ts n-1)^n || ts n-1..0
    t ← (tt n-1)^n || tt n-1..0
    p ← s * t
    return p 2n-1..0
endfunction mulx_s

function sat_s(tt, n, b)
    if tt n-1 = 0 and tt n-1..b-1 ≠ 0^n-b+1 then
        return 0^n-b+1 || 1^b-1
    endif
    if tt n-1 = 1 and tt n-1..b-1 ≠ 1^n-b+1 then
        return 1^n-b+1 || 0^b-1
    else
        return tt
    endif
```

```
endfunction sat_s

function q_madd(td, ts, tt, n)
   p ← mulx_s(ts, tt, n)
   d ← (td_{n-1} || td_{n-1..0} || 0^{n-1}) + p_{2n-1..0}
   d ← sat_s(d_{2n-1..n-1}, n+1, n)
   return d_{n-1..0}
endfunction q_madd
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception.




**Format:** `MADDR_Q.df`

`MADDR_Q.H wd,ws,wt` **MSA**

`MADDR_Q.W wd,ws,wt` **MSA**

**Purpose:** Vector Fixed-Point Multiply and Add Rounded

Vector fixed-point multiply and add rounded.

**Description:** `wd[i] ← saturate(round(wd[i] + ws[i] * wt[i]))`

The products of fixed-point elements in vector *wt* by fixed-point elements in vector *ws* are added to the fixed-point elements in vector *wd*. The multiplication result is not saturated, i.e. exact (-1) * (-1) = 1 is added to the destination. The rounded and saturated fixed-point results are stored back to *wd*.

Internally, the multiplication, addition, and rounding operate on data double the size of *df*. Truncation to fixed-point data format *df* is performed at the very last stage, after saturation.

The rounding is done by adding 1 to the most significant bit that is going to be discarded at truncation.

The operands and results are values in fixed-point data format *df*.

**Restrictions:**

No data-dependent exceptions are possible.

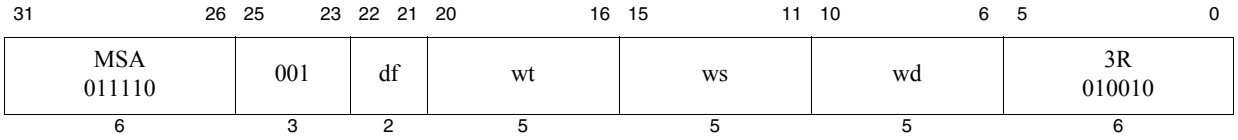
**Operation:**

```
MADDR_Q.H
    for i in 0 .. WRLEN/16-1
        WR[wd]16i+15..16i ←
                q_maddr(WR[wd]16i+15..16i, WR[ws]16i+15..16i, WR[wt]16i+15..16i, 16)
    endfor

MADDR_Q.W
    for i in 0 .. WRLEN/32-1
        WR[wd]32i+31..32i ←
                q_maddr(WR[wd]32i+31..32i, WR[ws]32i+31..32i, WR[wt]32i+31..32i, 32)
    endfor

function mulx_s(ts, tt, n)
    s ← (tsn-1)n || tsn-1..0
    t ← (ttn-1)n || ttn-1..0
    p ← s * t
    return p2n-1..0
endfunction mulx_s

function sat_s(tt, n, b)
    if ttn-1 = 0 and ttn-1..b-1 ≠ 0n-b+1 then
        return 0n-b+1 || 1b-1
    endif
    if ttn-1 = 1 and ttn-1..b-1 ≠ 1n-b+1 then
        return 1n-b+1 || 0b-1
    else
        return tt
```

```
    endif
endfunction sat_s

function q_maddr(td, ts, tt, n)
    p ← mulx_s(ts, tt, n)
    d ← (td_{n-1} || td_{n-1..0} || 0^{n-1}) + p_{2n-1..0}
    d ← d + (1 || 0^{n-2})
    d ← sat_s(d_{2n-1..n-1}, n+1, n)
    return d_{n-1..0}
endfunction q_maddr
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception.




**Format:** `MADDV.df`

`MADDV.B wd,ws,wt` **MSA**

`MADDV.H wd,ws,wt` **MSA**

`MADDV.W wd,ws,wt` **MSA**

`MADDV.D wd,ws,wt` **MSA**

**Purpose:** Vector Multiply and Add

Vector multiply and add.

**Description:** `wd[i] ← wd[i] + ws[i] * wt[i]`

The integer elements in vector *wt* are multiplied by integer elements in vector *ws* and added to the integer elements in vector *wd*. The most significant half of the multiplication result is discarded.

The operands and results are values in integer data format *df*.

**Restrictions:**

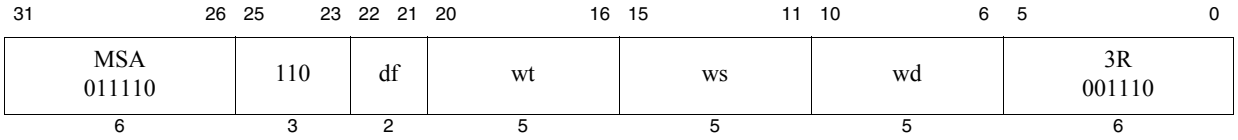
No data-dependent exceptions are possible.

**Operation:**

```
MADDV.B
    for i in 0 .. WRLEN/8-1
        WR[wd]8i+7..8i ←
                WR[wd]8i+7..8i + WR[ws]8i+7..8i * WR[wt]8i+7..8i
    endfor

MADDV.H
    for i in 0 .. WRLEN/16-1
        WR[wd]16i+15..16i ←
                WR[wd]16i+15..16i + WR[ws]16i+15..16i * WR[wt]16i+15..16i
    endfor

MADDV.W
    for i in 0 .. WRLEN/32-1
        WR[wd]32i+31..32i ←
                WR[wd]32i+31..32i + WR[ws]32i+31..32i * WR[wt]32i+31..32i
    endfor

MADDV.D
    for i in 0 .. WRLEN/64-1
        WR[wd]64i+63..64i ←
                WR[wd]64i+63..64i + WR[ws]64i+63..64i * WR[wt]64i+63..64i
    endfor
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception.




**Format:** `MAX_A.df`

`MAX_A.B wd,ws,wt` **MSA**

`MAX_A.H wd,ws,wt` **MSA**

`MAX_A.W wd,ws,wt` **MSA**

`MAX_A.D wd,ws,wt` **MSA**

**Purpose:** Vector Maximum Based on Absolute Values

Vector and vector maximum based on the absolute values.

**Description:** `wd[i] ← absolute_value(ws[i]) > absolute_value(wt[i])? ws[i]: wt[i]`

The value with the largest magnitude, i.e. absolute value, between corresponding signed elements in vector *ws* and vector *wt* are written to vector *wd*.

The minimum negative value representable has the largest absolute value.

The operands and results are values in integer data format *df*.

**Restrictions:**

No data-dependent exceptions are possible.

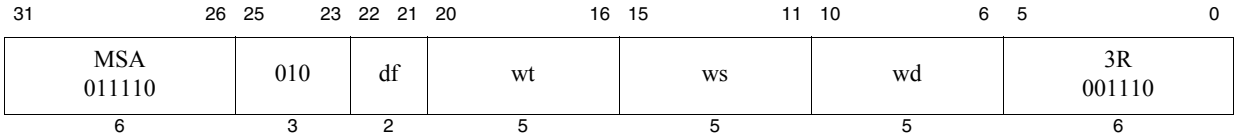
**Operation:**

```
MAX_A.B
    for i in 0 .. WRLEN/8-1
        WR[wd]8i+7..8i ← max_a(WR[ws]8i+7..8i, WR[wt]8i+7..8i, 8)
    endfor

MAX_A.H
    for i in 0 .. WRLEN/16-1
        WR[wd]16i+15..16i ← max_a(WR[ws]16i+15..16i, WR[wt]16i+15..16i, 16)
    endfor

MAX_A.W
    for i in 0 .. WRLEN/32-1
        WR[wd]32i+31..32i ← max_a(WR[ws]32i+31..32i, WR[wt]32i+31..32i, 32)
    endfor

MAX_A.D
    for i in 0 .. WRLEN/64-1
        WR[wd]64i+63..64i ← max_a(WR[ws]64i+63..64i, WR[wt]64i+63..64i, 64)
    endfor

function abs(tt, n)
    if ttn-1 = 1 then
        return -ttn-1...0
    else
        return ttn-1..0
    endif
endfunction abs
```

```
function max_a(ts, tt, n)
   t ← 0 || abs(tt, n)
   s ← 0 || abs(ts, n)
   if t < s then
      return ts
   else
      return tt
   endif
endfunction max_a
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception.




**Format:** `MAX_S.df`

`MAX_S.B wd,ws,wt` **MSA**

`MAX_S.H wd,ws,wt` **MSA**

`MAX_S.W wd,ws,wt` **MSA**

`MAX_S.D wd,ws,wt` **MSA**

**Purpose:** Vector Signed Maximum

Vector and vector signed maximum.

**Description:** `wd[i] ← max(ws[i], wt[i])`

Maximum values between signed elements in vector *wt* and signed elements in vector *ws* are written to vector *wd*.

The operands and results are values in integer data format *df*.

**Restrictions:**

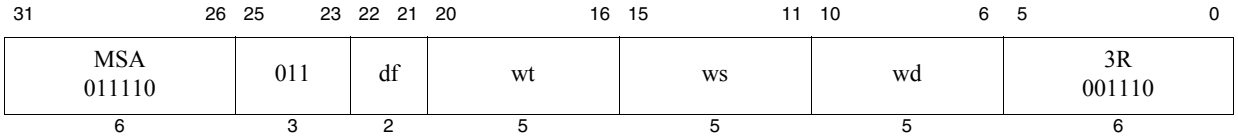
No data-dependent exceptions are possible.

**Operation:**

```
MAX_S.B
    for i in 0 .. WRLEN/8-1
        WR[wd]8i+7..8i ← max_s(WR[ws]8i+7..8i, WR[wt]8i+7..8i, 8)
    endfor

MAX_S.H
    for i in 0 .. WRLEN/16-1
        WR[wd]16i+15..16i ← max_s(WR[ws]16i+15..16i, WR[wt]16i+15..16i, 16)
    endfor

MAX_S.W
    for i in 0 .. WRLEN/32-1
        WR[wd]32i+31..32i ← max_s(WR[ws]32i+31..32i, WR[wt]32i+31..32i, 32)
    endfor

MAX_S.D
    for i in 0 .. WRLEN/64-1
        WR[wd]64i+63..64i ← max_s(WR[ws]64i+63..64i, WR[wt]64i+63..64i, 64)
    endfor

function max_s(ts, tt, n)
    t ← ttn-1 || tt
    s ← tsn-1 || ts
    if t < s then
        return ts
    else
        return tt
    endif
endfunction max_s
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception.




**Format:** `MAX_U.df`

`MAX_U.B wd,ws,wt` **MSA**

`MAX_U.H wd,ws,wt` **MSA**

`MAX_U.W wd,ws,wt` **MSA**

`MAX_U.D wd,ws,wt` **MSA**

**Purpose:** Vector Unsigned Maximum

Vector and vector unsigned maximum.

**Description:** `wd[i] ← max(ws[i], wt[i])`

Maximum values between unsigned elements in vector *wt* and unsigned elements in vector *ws* are written to vector *wd*.

The operands and results are values in integer data format *df*.

**Restrictions:**

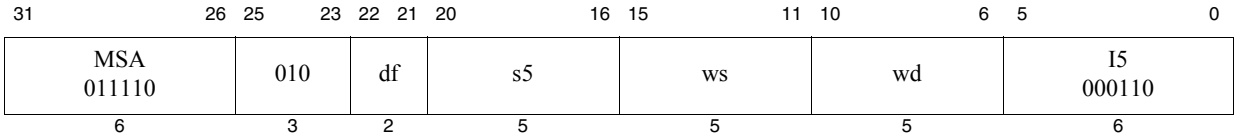
No data-dependent exceptions are possible.

**Operation:**

```
MAX_U.B
    for i in 0 .. WRLEN/8-1
        WR[wd]8i+7..8i ← max_u(WR[ws]8i+7..8i, WR[wt]8i+7..8i, 8)
    endfor

MAX_U.H
    for i in 0 .. WRLEN/16-1
        WR[wd]16i+15..16i ← max_u(WR[ws]16i+15..16i, WR[wt]16i+15..16i, 16)
    endfor

MAX_U.W
    for i in 0 .. WRLEN/32-1
        WR[wd]32i+31..32i ← max_u(WR[ws]32i+31..32i, WR[wt]32i+31..32i, 32)
    endfor

MAX_U.D
    for i in 0 .. WRLEN/64-1
        WR[wd]64i+63..64i ← max_u(WR[ws]64i+63..64i, WR[wt]64i+63..64i, 64)
    endfor

function max_u(ts, tt, n)
    t ← 0 || tt
    s ← 0 || ts
    if t < s then
        return ts
    else
        return tt
    endif
endfunction max_u
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception.




**Format:** `MAXI_S.df`

`MAXI_S.B wd,ws,s5` **MSA**

`MAXI_S.H wd,ws,s5` **MSA**

`MAXI_S.W wd,ws,s5` **MSA**

`MAXI_S.D wd,ws,s5` **MSA**

**Purpose:** Immediate Signed Maximum

Immediate and vector signed maximum.

**Description:** `wd[i] ← max(ws[i], s5)`

Maximum values between signed elements in vector *ws* and the 5-bit signed immediate *s5* are written to vector *wd*.

The operands and results are values in integer data format *df*.

**Restrictions:**

No data-dependent exceptions are possible.

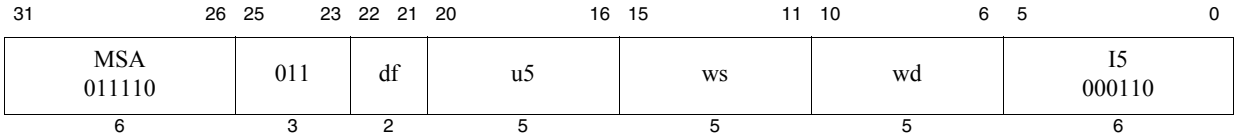
**Operation:**

```
MAXI_S.B
    t ← (s5_4)^3 || s5_4..0
    for i in 0 .. WRLEN/8-1
        WR[wd]_8i+7..8i ← max_s(WR[ws]_8i+7..8i, t, 8)
    endfor

MAXI_S.H
    t ← (s5_4)^11 || s5_4..0
    for i in 0 .. WRLEN/16-1
        WR[wd]_16i+15..16i ← max_s(WR[ws]_16i+15..16i, t, 16)
    endfor

MAXI_S.W
    t ← (s5_4)^27 || s5_4..0
    for i in 0 .. WRLEN/32-1
        WR[wd]_32i+31..32i ← max_s(WR[ws]_32i+31..32i, t, 32)
    endfor

MAXI_S.D
    t ← (s5_4)^59 || s5_4..0
    for i in 0 .. WRLEN/64-1
        WR[wd]_64i+63..64i ← max_s(WR[ws]_64i+63..64i, t, 64)
    endfor

function max_s(ts, tt, n)
    t ← tt_n-1 || tt
    s ← ts_n-1 || ts
    if t < s then
        return ts
    else
        return tt
```

```
    endif
endfunction max_s
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception.




**Format:** `MAXI_U.df`

`MAXI_U.B wd,ws,u5` **MSA**

`MAXI_U.H wd,ws,u5` **MSA**

`MAXI_U.W wd,ws,u5` **MSA**

`MAXI_U.D wd,ws,u5` **MSA**

**Purpose:** Immediate Unsigned Maximum

Immediate and vector unsigned maximum.

**Description:** `wd[i] ← max(ws[i], u5)`

Maximum values between unsigned elements in vector *ws* and the 5-bit unsigned immediate *u5* are written to vector *wd*.

The operands and results are values in integer data format *df*.

**Restrictions:**

No data-dependent exceptions are possible.

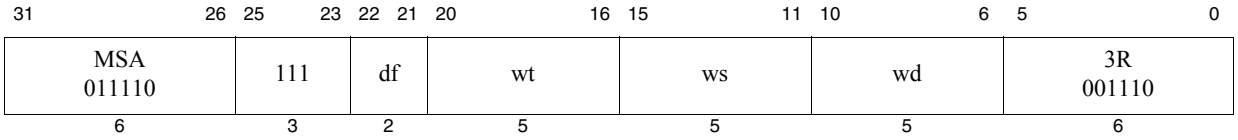
**Operation:**

```
MAXI_U.B
    t ← 0^3 || u5_4..0
    for i in 0 .. WRLEN/8-1
        WR[wd]_8i+7..8i ← max_u(WR[ws]_8i+7..8i, t, 8)
    endfor

MAXI_U.H
    t ← 0^11 || u5_4..0
    for i in 0 .. WRLEN/16-1
        WR[wd]_16i+15..16i ← max_u(WR[ws]_16i+15..16i, t, 16)
    endfor

MAXI_U.W
    t ← 0^27 || u5_4..0
    for i in 0 .. WRLEN/32-1
        WR[wd]_32i+31..32i ← max_u(WR[ws]_32i+31..32i, t, 32)
    endfor

MAXI_U.D
    t ← 0^59 || u5_4..0
    for i in 0 .. WRLEN/64-1
        WR[wd]_64i+63..64i ← max_u(WR[ws]_64i+63..64i, t, 64)
    endfor

function max_u(ts, tt, n)
    t ← 0 || tt
    s ← 0 || ts
    if t < s then
        return ts
    else
```

```
        return tt
      endif
    endfunction max_u
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception.




**Format:** `MIN_A.df`

`MIN_A.B wd,ws,wt` **MSA**

`MIN_A.H wd,ws,wt` **MSA**

`MIN_A.W wd,ws,wt` **MSA**

`MIN_A.D wd,ws,wt` **MSA**

**Purpose:** Vector Minimum Based on Absolute Value

Vector and vector minimum based on the absolute values.

**Description:** `wd[i] ← absolute_value(ws[i]) < absolute_value(wt[i])? ws[i]: wt[i]`

The value with the smallest magnitude, i.e. absolute value, between corresponding signed elements in vector *ws* and vector *wt* are written to vector *wd*.

The minimum negative value representable has the largest absolute value.

The operands and results are values in integer data format *df*.

**Restrictions:**

No data-dependent exceptions are possible.

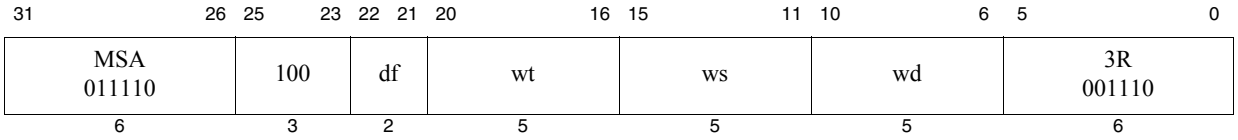
**Operation:**

```
MIN_A.B
    for i in 0 .. WRLEN/8-1
        WR[wd]8i+7..8i ← min_a(WR[ws]8i+7..8i, WR[wt]8i+7..8i, 8)
    endfor

MIN_A.H
    for i in 0 .. WRLEN/16-1
        WR[wd]16i+15..16i ← min_a(WR[ws]16i+15..16i, WR[wt]16i+15..16i, 16)
    endfor

MIN_A.W
    for i in 0 .. WRLEN/32-1
        WR[wd]32i+31..32i ← min_a(WR[ws]32i+31..32i, WR[wt]32i+31..32i, 32)
    endfor

MIN_A.D
    for i in 0 .. WRLEN/64-1
        WR[wd]64i+63..64i ← min_a(WR[ws]64i+63..64i, WR[wt]64i+63..64i, 64)
    endfor

function min_a(ts, tt, n)
    t ← 0 || abs(tt, n)
    s ← 0 || abs(ts, n)
    if t > s then
        return ts
    else
        return tt
    endif
```

```
endfunction min_a
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception.




**Format:** `MIN_S.df`

`MIN_S.B wd,ws,wt` **MSA**

`MIN_S.H wd,ws,wt` **MSA**

`MIN_S.W wd,ws,wt` **MSA**

`MIN_S.D wd,ws,wt` **MSA**

**Purpose:** Vector Signed Minimum

Vector and vector signed minimum.

**Description:** `wd[i] ← min(ws[i], wt[i])`

Minimum values between signed elements in vector *wt* and signed elements in vector *ws* are written to vector *wd*.

The operands and results are values in integer data format *df*.

**Restrictions:**

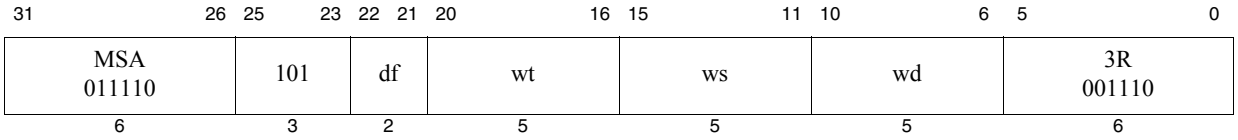
No data-dependent exceptions are possible.

**Operation:**

```
MIN_S.B
    for i in 0 .. WRLEN/8-1
        WR[wd]8i+7..8i ← min_s(WR[ws]8i+7..8i, WR[wt]8i+7..8i, 8)
    endfor

MIN_S.H
    for i in 0 .. WRLEN/16-1
        WR[wd]16i+15..16i ← min_s(WR[ws]16i+15..16i, WR[wt]16i+15..16i, 16)
    endfor

MIN_S.W
    for i in 0 .. WRLEN/32-1
        WR[wd]32i+31..32i ← min_s(WR[ws]32i+31..32i, WR[wt]32i+31..32i, 32)
    endfor

MIN_S.D
    for i in 0 .. WRLEN/64-1
        WR[wd]64i+63..64i ← min_s(WR[ws]64i+63..64i, WR[wt]64i+63..64i, 64)
    endfor

function min_s(ts, tt, n)
    t ← ttn-1 || tt
    s ← tsn-1 || ts
    if t > s then
        return ts
    else
        return tt
    endif
endfunction min_s
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception.




**Format:** `MIN_U.df`

`MIN_U.B wd,ws,wt` **MSA**

`MIN_U.H wd,ws,wt` **MSA**

`MIN_U.W wd,ws,wt` **MSA**

`MIN_U.D wd,ws,wt` **MSA**

**Purpose:** Vector Unsigned Minimum

Vector and vector unsigned minimum.

**Description:** `wd[i] ← min(ws[i], wt[i])`

Minimum values between unsigned elements in vector *wt* and unsigned elements in vector *ws* are written to vector *wd*.

The operands and results are values in integer data format *df*.

**Restrictions:**

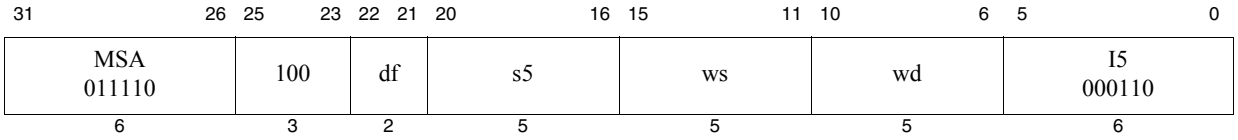
No data-dependent exceptions are possible.

**Operation:**

```
MIN_U.B
    for i in 0 .. WRLEN/8-1
        WR[wd]8i+7..8i ← min_u(WR[ws]8i+7..8i, WR[wt]8i+7..8i, 8)
    endfor

MIN_U.H
    for i in 0 .. WRLEN/16-1
        WR[wd]16i+15..16i ← min_u(WR[ws]16i+15..16i, WR[wt]16i+15..16i, 16)
    endfor

MIN_U.W
    for i in 0 .. WRLEN/32-1
        WR[wd]32i+31..32i ← min_u(WR[ws]32i+31..32i, WR[wt]32i+31..32i, 32)
    endfor

MIN_U.D
    for i in 0 .. WRLEN/64-1
        WR[wd]64i+63..64i ← min_u(WR[ws]64i+63..64i, WR[wt]64i+63..64i, 64)
    endfor

function min_u(ts, tt, n)
    t ← 0 || tt
    s ← 0 || ts
    if t > s then
        return ts
    else
        return tt
    endif
endfunction min_u
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception.




**Format:** `MINI_S.df`

`MINI_S.B wd,ws,s5` **MSA**

`MINI_S.H wd,ws,s5` **MSA**

`MINI_S.W wd,ws,s5` **MSA**

`MINI_S.D wd,ws,s5` **MSA**

**Purpose:** Immediate Signed Minimum

Immediate and vector signed minimum.

**Description:** `wd[i] ← min(ws[i], s5)`

Minimum values between signed elements in vector *ws* and the 5-bit signed immediate *s5* are written to vector *wd*.

The operands and results are values in integer data format *df*.

**Restrictions:**

No data-dependent exceptions are possible.

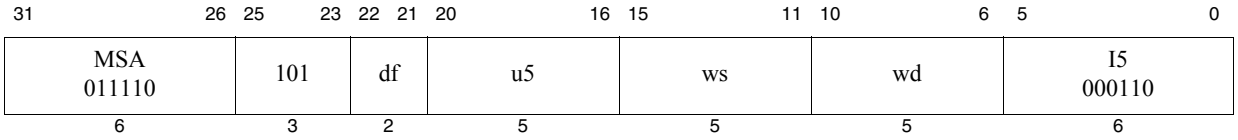
**Operation:**

```
MINI_S.B
    t ← (s5_4)^3 || s5_4..0
    for i in 0 .. WRLEN/8-1
        WR[wd]_8i+7..8i ← min_s(WR[ws]_8i+7..8i, t, 8)
    endfor

MINI_S.H
    t ← (s5_4)^11 || s5_4..0
    for i in 0 .. WRLEN/16-1
        WR[wd]_16i+15..16i ← min_s(WR[ws]_16i+15..16i, t, 16)
    endfor

MINI_S.W
    t ← (s5_4)^27 || s5_4..0
    for i in 0 .. WRLEN/32-1
        WR[wd]_32i+31..32i ← min_s(WR[ws]_32i+31..32i, t, 32)
    endfor

MINI_S.D
    t ← (s5_4)^59 || s5_4..0
    for i in 0 .. WRLEN/64-1
        WR[wd]_64i+63..64i ← min_s(WR[ws]_64i+63..64i, t, 64)
    endfor

function min_s(ts, tt, n)
    t ← tt_n-1 || tt
    s ← ts_n-1 || ts
    if t > s then
        return ts
    else
        return tt
```

```
    endif
endfunction min_s
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception.




**Format:** `MINI_U.df`

`MINI_U.B wd,ws,u5` **MSA**

`MINI_U.H wd,ws,u5` **MSA**

`MINI_U.W wd,ws,u5` **MSA**

`MINI_U.D wd,ws,u5` **MSA**

**Purpose:** Immediate Unsigned Minimum

Immediate and vector unsigned minimum.

**Description:** `wd[i] ← min(ws[i], u5)`

Minimum values between unsigned elements in vector *ws* and the 5-bit unsigned immediate *u5* are written to vector *wd*.

The operands and results are values in integer data format *df*.

**Restrictions:**

No data-dependent exceptions are possible.

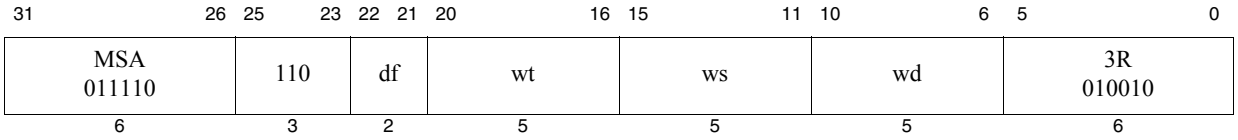
**Operation:**

```
MINI_U.B
    t ← 0^3 || u5_4..0
    for i in 0 .. WRLEN/8-1
        WR[wd]_8i+7..8i ← min_u(WR[ws]_8i+7..8i, t, 8)
    endfor

MINI_U.H
    t ← 0^11 || u5_4..0
    for i in 0 .. WRLEN/16-1
        WR[wd]_16i+15..16i ← min_u(WR[ws]_16i+15..16i, t, 16)
    endfor

MINI_U.W
    t ← 0^27 || u5_4..0
    for i in 0 .. WRLEN/32-1
        WR[wd]_32i+31..32i ← min_u(WR[ws]_32i+31..32i, t, 32)
    endfor

MINI_U.D
    t ← 0^59 || u5_4..0
    for i in 0 .. WRLEN/64-1
        WR[wd]_64i+63..64i ← min_u(WR[ws]_64i+63..64i, t, 64)
    endfor

function min_u(ts, tt, n)
    t ← 0 || tt
    s ← 0 || ts
    if t > s then
        return ts
    else
```

```
        return tt
    endif
endfunction min_u
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception.




**Format:** `MOD_S.df`

`MOD_S.B wd,ws,wt` **MSA**

`MOD_S.H wd,ws,wt` **MSA**

`MOD_S.W wd,ws,wt` **MSA**

`MOD_S.D wd,ws,wt` **MSA**

**Purpose:** Vector Signed Modulo

Vector signed remainder (modulo).

**Description:** `wd[i] ← ws[i] mod wt[i]`

The signed integer elements in vector *ws* are divided by signed integer elements in vector *wt*. The remainder of the same sign as the dividend is written to vector *wd*. If a divisor element vector *wt* is zero, the result value is **UNPREDICTABLE**.

The operands and results are values in integer data format *df*.

**Restrictions:**

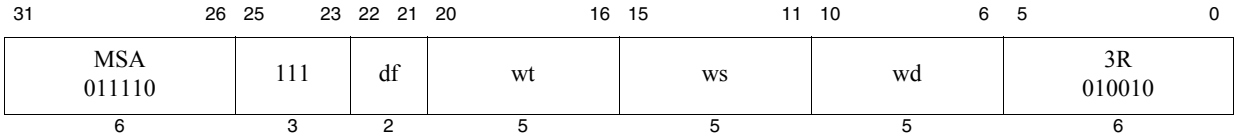
No data-dependent exceptions are possible.

**Operation:**

```
MOD_S.B
    for i in 0 .. WRLEN/8-1
        WR[wd]8i+7..8i ← WR[ws]8i+7..8i mod WR[wt]8i+7..8i
    endfor

MOD_S.H
    for i in 0 .. WRLEN/16-1
        WR[wd]16i+15..16i ← WR[ws]16i+15..16i mod WR[wt]16i+15..16i
    endfor

MOD_S.W
    for i in 0 .. WRLEN/32-1
        WR[wd]32i+31..32i ← WR[ws]32i+31..32i mod WR[wt]32i+31..32i
    endfor

MOD_S.D
    for i in 0 .. WRLEN/64-1
        WR[wd]64i+63..64i ← WR[ws]64i+63..64i mod WR[wt]64i+63..64i
    endfor
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception.




**Format:** `MOD_U.df`

`MOD_U.B wd,ws,wt` **MSA**

`MOD_U.H wd,ws,wt` **MSA**

`MOD_U.W wd,ws,wt` **MSA**

`MOD_U.D wd,ws,wt` **MSA**

**Purpose:** Vector Unsigned Modulo

Vector unsigned remainder (modulo).

**Description:** `wd[i] ← ws[i] umod wt[i]`

The unsigned integer elements in vector *ws* are divided by unsigned integer elements in vector *wt*. The remainder is written to vector *wd*. If a divisor element vector *wt* is zero, the result value is **UNPREDICTABLE**.

The operands and results are values in integer data format *df*.

**Restrictions:**

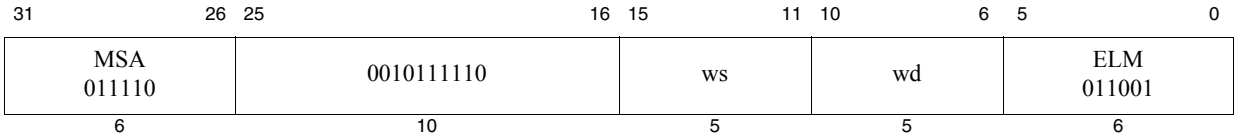
No data-dependent exceptions are possible.

**Operation:**

```
MOD_U.B
    for i in 0 .. WRLEN/8-1
        WR[wd]8i+7..8i ← WR[ws]8i+7..8i umod WR[wt]8i+7..8i
    endfor

MOD_U.H
    for i in 0 .. WRLEN/16-1
        WR[wd]16i+15..16i ← WR[ws]16i+15..16i umod WR[wt]16i+15..16i
    endfor

MOD_U.W
    for i in 0 .. WRLEN/32-1
        WR[wd]32i+31..32i ← WR[ws]32i+31..32i umod WR[wt]32i+31..32i
    endfor

MOD_U.D
    for i in 0 .. WRLEN/64-1
        WR[wd]64i+63..64i ← WR[ws]64i+63..64i umod WR[wt]64i+63..64i
    endfor
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception.




**Format:** `MOVE.V`
`MOVE.V wd,ws` **MSA**

**Purpose:** Vector Move

Vector to vector move.

**Description:** `wd ← ws`

Copy all WRLEN bits in vector *ws* to vector *wd*.

The operand and result are bit vector values.

**Restrictions:**

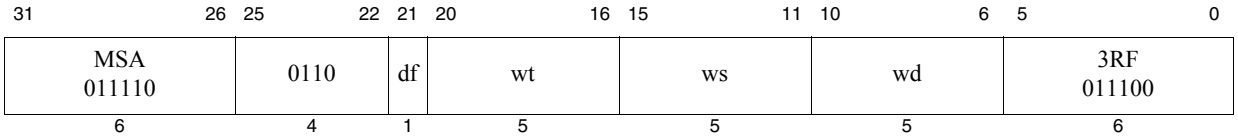
No data-dependent exceptions are possible.

**Operation:**

```
WR[wd] ← WR[ws]
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception.




**Format:** `MSUB_Q.df`

`MSUB_Q.H wd,ws,wt` **MSA**

`MSUB_Q.W wd,ws,wt` **MSA**

**Purpose:** Vector Fixed-Point Multiply and Subtract

Vector fixed-point multiply and subtract.

**Description:** `wd[i] ← saturate(wd[i] - ws[i] * wt[i])`

The product of fixed-point elements in vector *wt* by fixed-point elements in vector *ws* are subtracted from the fixed-point elements in vector *wd*. The multiplication result is not saturated, i.e. exact (-1) * (-1) = 1 is subtracted from the destination. The saturated fixed-point results are stored back to *wd*.

Internally, the multiplication and subtraction operate on data double the size of *df*. Truncation to fixed-point data format *df* is performed at the very last stage, after saturation.

The operands and results are values in fixed-point data format *df*.

**Restrictions:**

No data-dependent exceptions are possible.

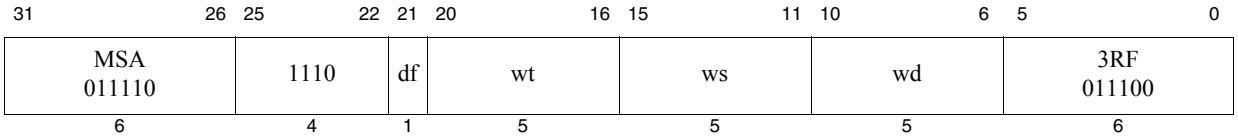
**Operation:**

```
MSUB_Q.H
    for i in 0 .. WRLEN/16-1
        WR[wd]16i+15..16i ←
                q_msub(WR[wd]16i+15..16i, WR[ws]16i+15..16i, WR[wt]16i+15..16i, 16)
    endfor

MSUB_Q.W
    for i in 0 .. WRLEN/32-1
        WR[wd]32i+31..32i ←
                q_msub(WR[wd]32i+31..32i, WR[ws]32i+31..32i, WR[wt]32i+31..32i, 32)
    endfor

function mulx_s(ts, tt, n)
    s ← (ts(n-1))^n || tsn-1..0
    t ← (tt(n-1))^n || ttn-1..0
    p ← s * t
    return p2n-1..0
endfunction mulx_s

function sat_s(tt, n, b)
    if ttn-1 = 0 and ttn-1..b-1 ≠ 0^(n-b+1) then
        return 0^(n-b+1) || 1^(b-1)
    endif
    if ttn-1 = 1 and ttn-1..b-1 ≠ 1^(n-b+1) then
        return 1^(n-b+1) || 0^(b-1)
    else
        return tt
    endif
```

```
endfunction sat_s

function q_msub(td, ts, tt, n)
   p ← mulx_s(ts, tt, n)
   d ← (td_{n-1} || td_{n-1..0} || 0^{n-1}) - p_{2n-1..0}
   d ← sat_s(d_{2n-1..n-1}, n+1, n)
   return d_{n-1..0}
endfunction q_msub
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception.




**Format:** `MSUBR_Q.df`

`MSUBR_Q.H wd,ws,wt` **MSA**

`MSUBR_Q.W wd,ws,wt` **MSA**

**Purpose:** Vector Fixed-Point Multiply and Subtract Rounded

Vector fixed-point multiply and subtract rounded.

**Description:** `wd[i] ← saturate(round(wd[i] - ws[i] * wt[i]))`

The products of fixed-point elements in vector *wt* by fixed-point elements in vector *ws* are subtracted from the fixed-point elements in vector *wd*. The multiplication result is not saturated, i.e. exact (-1) * (-1) = 1 is subtracted from the destination. The rounded and saturated fixed-point results are stored back to *wd*.

Internally, the multiplication, subtraction, and rounding operate on data double the size of *df*. Truncation to fixed-point data format *df* is performed at the very last stage, after saturation.

The rounding is done by adding 1 to the most significant bit that is going to be discarded at truncation.

The operands and results are values in fixed-point data format *df*.

**Restrictions:**

No data-dependent exceptions are possible.

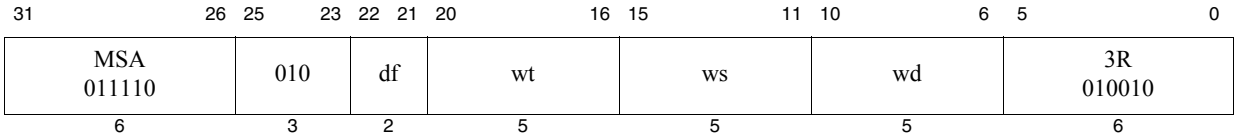
**Operation:**

```
MSUBR_Q.H
    for i in 0 .. WRLEN/16-1
        WR[wd]16i+15..16i ←
                q_msubr(WR[wd]16i+15..16i, WR[ws]16i+15..16i, WR[wt]16i+15..16i, 16)
    endfor

MSUBR_Q.W
    for i in 0 .. WRLEN/32-1
        WR[wd]32i+31..32i ←
                q_msubr(WR[wd]32i+31..32i, WR[ws]32i+31..32i, WR[wt]32i+31..32i, 32)
    endfor

function mulx_s(ts, tt, n)
    s ← (tsn-1)n || tsn-1..0
    t ← (ttn-1)n || ttn-1..0
    p ← s * t
    return p2n-1..0
endfunction mulx_s

function sat_s(tt, n, b)
    if ttn-1 = 0 and ttn-1..b-1 ≠ 0n-b+1 then
        return 0n-b+1 || 1b-1
    endif
    if ttn-1 = 1 and ttn-1..b-1 ≠ 1n-b+1 then
        return 1n-b+1 || 0b-1
    else
        return tt
```

```
    endif
endfunction sat_s

function q_msubr(td, ts, tt, n)
    p ← mulx_s(ts, tt, n)
    d ← (td_{n-1} || td_{n-1..0} || 0^{n-1}) - p_{2n-1..0}
    d ← d + (1 || 0^{n-2})
    d ← sat_s(d_{2n-1..n-1}, n+1, n)
    return d_{n-1..0}
endfunction q_msubr
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception.




**Format:** `MSUBV.df`

`MSUBV.B wd,ws,wt` **MSA**

`MSUBV.H wd,ws,wt` **MSA**

`MSUBV.W wd,ws,wt` **MSA**

`MSUBV.D wd,ws,wt` **MSA**

**Purpose:** Vector Multiply and Subtract

Vector multiply and subtract.

**Description:** `wd[i] ← wd[i] - ws[i] * wt[i]`

The integer elements in vector *wt* are multiplied by integer elements in vector *ws* and subtracted from the integer elements in vector *wd*. The most significant half of the multiplication result is discarded.

The operands and results are values in integer data format *df*.

**Restrictions:**

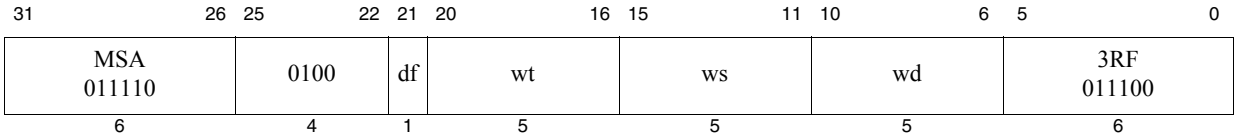
No data-dependent exceptions are possible.

**Operation:**

```
MSUBV.B
    for i in 0 .. WRLEN/8-1
        WR[wd]8i+7..8i ←
                WR[wd]8i+7..8i - WR[ws]8i+7..8i * WR[wt]8i+7..8i
    endfor

MSUBV.H
    for i in 0 .. WRLEN/16-1
        WR[wd]16i+15..16i ←
                WR[wd]16i+15..16i - WR[ws]16i+15..16i * WR[wt]16i+15..16i
    endfor

MSUBV.W
    for i in 0 .. WRLEN/32-1
        WR[wd]32i+31..32i ←
                WR[wd]32i+31..32i - WR[ws]32i+31..32i * WR[wt]32i+31..32i
    endfor

MSUBV.D
    for i in 0 .. WRLEN/64-1
        WR[wd]64i+63..64i ←
                WR[wd]64i+63..64i - WR[ws]64i+63..64i * WR[wt]64i+63..64i
    endfor
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception.




**Format:** `MUL_Q.df`

`MUL_Q.H wd,ws,wt` **MSA**

`MUL_Q.W wd,ws,wt` **MSA**

**Purpose:** Vector Fixed-Point Multiply

Vector fixed-point multiplication.

**Description:** `wd[i] ← ws[i] * wt[i]`

The fixed-point elements in vector *wt* multiplied by fixed-point elements in vector *ws*. The result is written to vector *wd*.

Fixed-point multiplication for 16-bit Q15 and 32-bit Q31 is a regular signed multiplication followed by one bit shift left with saturation. Only the most significant half of the result is preserved.

The operands and results are values in fixed-point data format *df*.

**Restrictions:**

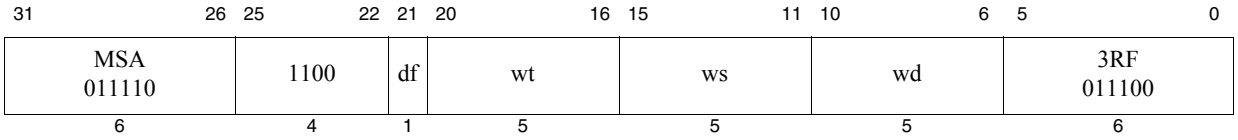
No data-dependent exceptions are possible.

**Operation:**

```
MUL_Q.H
    for i in 0 .. WRLEN/16-1
        WR[wd]16i+15..16i ← q_mul(WR[ws]16i+15..16i, WR[wt]16i+15..16i, 16)
    endfor

MUL_Q.W
    for i in 0 .. WRLEN/32-1
        WR[wd]32i+31..32i ← q_mul(WR[ws]32i+31..32i, WR[wt]32i+31..32i, 32)
    endfor

function mulx_s(ts, tt, n)
    s ← (tsn-1)n || tsn-1..0
    t ← (ttn-1)n || ttn-1..0
    p ← s * t
    return p2n-1..0
endfunction mulx_s

function q_mul(ts, tt, n)
    if ts = 1 || 0n-1 and tt = 1 || 0n-1 then
        return 0 || 1n-1
    else
        p ← mulx_s(ts, tt, n)
        return p2n-2..n-1
    endif
endfunction q_mul
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception.




**Format:** `MULR_Q.df`

`MULR_Q.H wd,ws,wt` **MSA**

`MULR_Q.W wd,ws,wt` **MSA**

**Purpose:** Vector Fixed-Point Multiply Rounded

Vector fixed-point multiply rounded.

**Description:** `wd[i] ← round(ws[i] * wt[i])`

The fixed-point elements in vector *wt* multiplied by fixed-point elements in vector *ws*. The rounded result is written to vector *wd*.

Fixed-point multiplication for 16-bit Q15 and 32-bit Q31 is a regular signed multiplication followed by one bit shift left with saturation. Only the most significant half of the result is preserved.

The rounding is done by adding 1 to the most significant bit that is going to be discarded prior to shifting left the full multiplication result.

The operands and results are values in fixed-point data format *df*.

**Restrictions:**

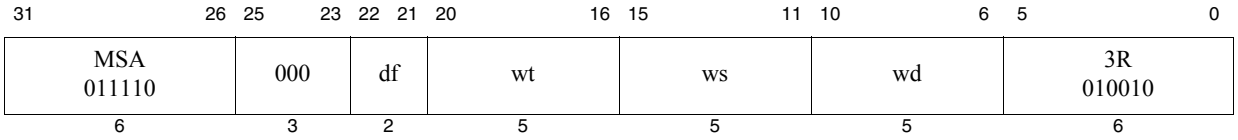
No data-dependent exceptions are possible.

**Operation:**

```
MULR_Q.H
   for i in 0 .. WRLEN/16-1
      WR[wd]16i+15..16i ← q_mulr(WR[ws]16i+15..16i, WR[wt]16i+15..16i, 16)
   endfor

MULR_Q.W
   for i in 0 .. WRLEN/32-1
      WR[wd]32i+31..32i ← q_mulr(WR[ws]32i+31..32i, WR[wt]32i+31..32i, 32)
   endfor

function mulx_s(ts, tt, n)
   s ← (tsn-1)n || tsn-1..0
   t ← (ttn-1)n || ttn-1..0
   p ← s * t
   return p2n-1..0
endfunction mulx_s

function q_mulr(ts, tt, n)
   if ts = 1 || 0n-1 and tt = 1 || 0n-1 then
      return 0 || 1n-1
   else
      p ← mulx_s(ts, tt, n)
      p ← p + (1 || 0n-2)
      return p2n-2..n-1
endfunction q_mulr
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception.




**Format:** `MULV.df`

`MULV.B wd,ws,wt` **MSA**

`MULV.H wd,ws,wt` **MSA**

`MULV.W wd,ws,wt` **MSA**

`MULV.D wd,ws,wt` **MSA**

**Purpose:** Vector Multiply

Vector multiply.

**Description:** `wd[i] ← ws[i] * wt[i]`

The integer elements in vector *wt* are multiplied by integer elements in vector *ws*. The result is written to vector *wd*. The most significant half of the multiplication result is discarded.

The operands and results are values in integer data format *df*.

**Restrictions:**

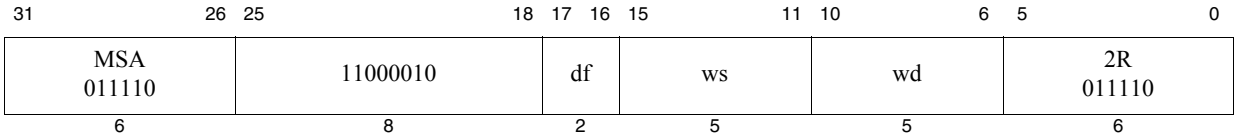
No data-dependent exceptions are possible.

**Operation:**

```
MULV.B
    for i in 0 .. WRLEN/8-1
        WR[wd]8i+7..8i ← WR[ws]8i+7..8i * WR[wt]8i+7..8i
    endfor

MULV.H
    for i in 0 .. WRLEN/16-1
        WR[wd]16i+15..16i ← WR[ws]16i+15..16i * WR[wt]16i+15..16i
    endfor

MULV.W
    for i in 0 .. WRLEN/32-1
        WR[wd]32i+31..32i ← WR[ws]32i+31..32i * WR[wt]32i+31..32i
    endfor

MULV.D
    for i in 0 .. WRLEN/64-1
        WR[wd]64i+63..64i ← WR[ws]64i+63..64i * WR[wt]64i+63..64i
    endfor
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception.




**Format:** `NLOC.df`

`NLOC.B wd,ws` **MSA**

`NLOC.H wd,ws` **MSA**

`NLOC.W wd,ws` **MSA**

`NLOC.D wd,ws` **MSA**

**Purpose:** Vector Leading Ones Count

Vector element count of leading bits set to 1.

**Description:** `wd[i] ← leading_one_count(ws[i])`

The number of leading ones for elements in vector *ws* is stored to the elements in vector *wd*.

The operands and results are values in integer data format *df*.

**Restrictions:**

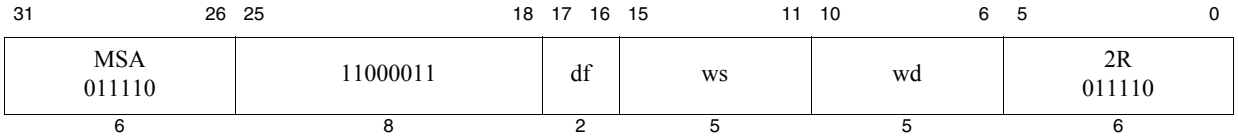
No data-dependent exceptions are possible.

**Operation:**

```
NLOC.B
    for i in 0 .. WRLEN/8-1
        WR[wd]8i+7..8i ← leading_one_count(WR[ws]8i+7..8i, 8)
    endfor

NLOC.H
    for i in 0 .. WRLEN/16-1
        WR[wd]16i+15..16i ← leading_one_count(WR[ws]16i+15..16i, 16)
    endfor

NLOC.W
    for i in 0 .. WRLEN/32-1
        WR[wd]32i+31..32i ← leading_one_count(WR[ws]32i+31..32i, 32)
    endfor

NLOC.D
    for i in 0 .. WRLEN/64-1
        WR[wd]64i+63..64i ← leading_one_count(WR[ws]64i+63..64i, 64)
    endfor

function leading_one_count(tt, n)
    z ← 0
    for i in n-1..0
        if tti = 0 then
            return z
        else
            z ← z + 1
        endif
endfunction leading_one_count
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception.




**Format:** `NLZC.df`

`NLZC.B wd,ws` **MSA**

`NLZC.H wd,ws` **MSA**

`NLZC.W wd,ws` **MSA**

`NLZC.D wd,ws` **MSA**

**Purpose:** Vector Leading Zeros Count

Vector element count of leading bits set to 0.

**Description:** `wd[i] ← leading_zero_count(ws[i])`

The number of leading zeroes for elements in vector *ws* is stored to the elements in vector *wd*.

The operands and results are values in integer data format *df*.

**Restrictions:**

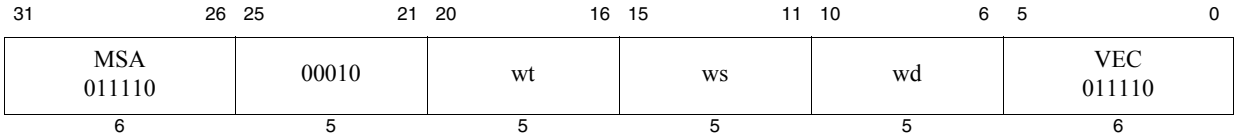
No data-dependent exceptions are possible.

**Operation:**

```
NLZC.B
    for i in 0 .. WRLEN/8-1
        WR[wd]8i+7..8i ← leading_zero_count(WR[ws]8i+7..8i, 8)
    endfor

NLZC.H
    for i in 0 .. WRLEN/16-1
        WR[wd]16i+15..16i ← leading_zero_count(WR[ws]16i+15..16i, 16)
    endfor

NLZC.W
    for i in 0 .. WRLEN/32-1
        WR[wd]32i+31..32i ← leading_zero_count(WR[ws]32i+31..32i, 32)
    endfor

NLZC.D
    for i in 0 .. WRLEN/64-1
        WR[wd]64i+63..64i ← leading_zero_count(WR[ws]64i+63..64i, 64)
    endfor

function leading_zero_count(tt, n)
    z ← 0
    for i in n-1..0
        if tti = 1 then
            return z
        else
            z ← z + 1
        endif
endfunction leading_zero_count
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception.




**Format:** `NOR.V`
`NOR.V wd,ws,wt` **MSA**

**Purpose:** Vector Logical Negated Or

Vector by vector logical negated or.

**Description:** `wd ← ws NOR wt`

Each bit of vector *ws* is combined with the corresponding bit of vector *wt* in a bitwise logical NOR operation. The result is written to vector *wd*.

The operands and results are bit vector values.

**Restrictions:**

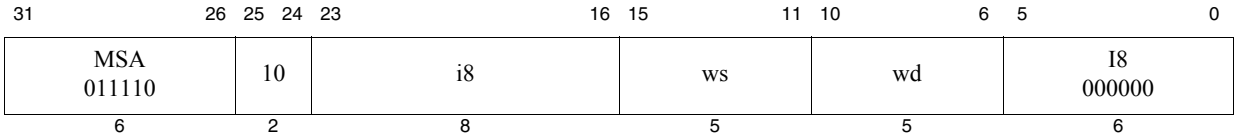
No data-dependent exceptions are possible.

**Operation:**

```
WR[wd] ← WR[ws] nor WR[wt]
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception.




**Format:** `NORI.B`
`NORI.B wd,ws,i8` **MSA**

**Purpose:** Immediate Logical Negated Or

Immediate by vector logical negated or.

**Description:** `wd[i] ← ws[i] NOR i8`

Each byte element of vector *ws* is combined with the 8-bit immediate *i8* in a bitwise logical NOR operation. The result is written to vector *wd*.

The operands and results are values in integer byte data format.

**Restrictions:**

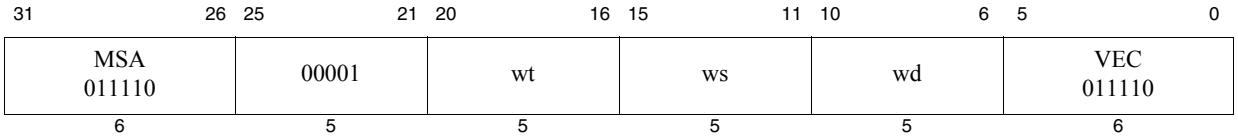
No data-dependent exceptions are possible.

**Operation:**

```
for i in 0 .. WRLEN/8-1
  WR[wd]8i+7..8i ← WR[ws]8i+7..8i nor i87..0
endfor
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception.




**Format:** `OR.V`
`OR.V wd,ws,wt` **MSA**

**Purpose:** Vector Logical Or

Vector by vector logical or.

**Description:** `wd ← ws OR wt`

Each bit of vector *ws* is combined with the corresponding bit of vector *wt* in a bit wise logical OR operation. The result is written to vector *wd*.

The operands and results are bit vector values.

**Restrictions:**

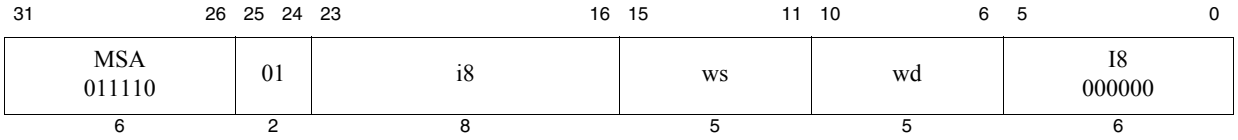
No data-dependent exceptions are possible.

**Operation:**

```
WR[wd] ← WR[ws] or WR[wt]
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception.




**Format:** `ORI.B`
`ORI.B wd,ws,i8` **MSA**

**Purpose:** Immediate Logical Or

Immediate by vector logical or.

**Description:** `wd[i] ← ws[i] OR i8`

Each byte element of vector *ws* is combined with the 8-bit immediate *i8* in a bitwise logical OR operation. The result is written to vector *wd*.

The operands and results are values in integer byte data format.

**Restrictions:**

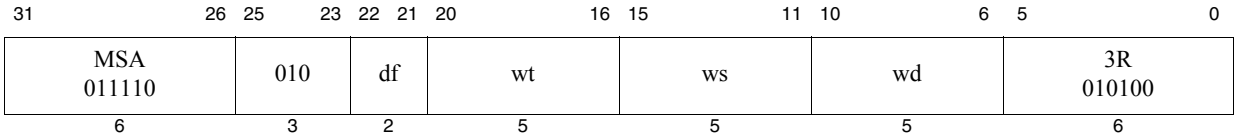
No data-dependent exceptions are possible.

**Operation:**

```
for i in 0 .. WRLEN/8-1
  WR[wd]8i+7..8i ← WR[ws]8i+7..8i or i87..0
endfor
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception.




**Format:** `PCKEV.df`

`PCKEV.B wd,ws,wt` **MSA**

`PCKEV.H wd,ws,wt` **MSA**

`PCKEV.W wd,ws,wt` **MSA**

`PCKEV.D wd,ws,wt` **MSA**

**Purpose:** Vector Pack Even

Vector even elements copy.

**Description:** `left_half(wd)[i] ← ws[2i]; right_half(wd)[i] ← wt[2i]`

Even elements in vector *ws* are copied to the left half of vector *wd* and even elements in vector *wt* are copied to the right half of vector *wd*.

The operands and results are values in integer data format *df*.

**Restrictions:**

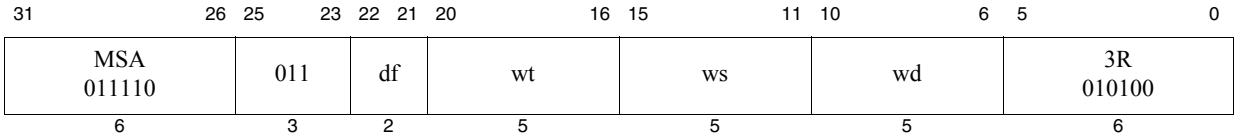
No data-dependent exceptions are possible.

**Operation:**

```
PCKEV.B
   for i in 0 .. WRLEN/16-1
      j ← 2 * i
      WR[wd]8i+7+WRLEN/2..8i+WRLEN/2 ← WR[ws]8j+7..8j
      WR[wd]8i+7..8i ← WR[wt]8j+7..8j
   endfor

PCKEV.H
   for i in 0 .. WRLEN/32-1
      j ← 2 * i
      WR[wd]16i+15+WRLEN/2..16j+WRLEN/2 ← WR[ws]16j+15..16j
      WR[wd]16i+15..16i ← WR[wt]16j+15..16j
   endfor

PCKEV.W
   for i in 0 .. WRLEN/64-1
      j ← 2 * i
      WR[wd]32i+31+WRLEN/2..32j+WRLEN/2 ← WR[ws]32j+31..32j
      WR[wd]32i+31..32i ← WR[wt]32j+31..32j
   endfor

PCKEV.D
   for i in 0 .. WRLEN/128-1
      j ← 2 * i
      WR[wd]64i+63+WRLEN/2..64j+WRLEN/2 ← WR[ws]64j+63..64j
      WR[wd]64i+63..64i ← WR[wt]64j+63..64j
   endfor
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception.




**Format:** `PCKOD.df`

`PCKOD.B wd,ws,wt` **MSA**

`PCKOD.H wd,ws,wt` **MSA**

`PCKOD.W wd,ws,wt` **MSA**

`PCKOD.D wd,ws,wt` **MSA**

**Purpose:** Vector Pack Odd

Vector odd elements copy.

**Description:** `left_half(wd)[i] ← ws[2i+1]; right_half(wd)[i] ← wt[2i+1]`

Odd elements in vector *ws* are copied to the left half of vector *wd* and odd elements in vector *wt* are copied to the right half of vector *wd*.

The operands and results are values in integer data format *df*.

**Restrictions:**

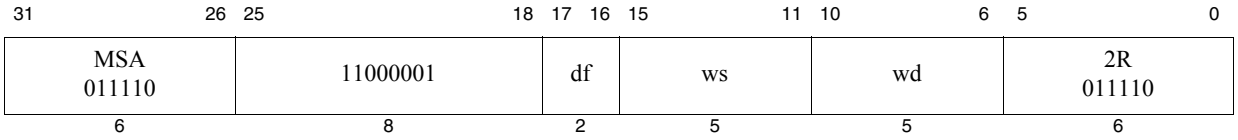
No data-dependent exceptions are possible.

**Operation:**

```
PCKOD.B
   for i in 0 .. WRLEN/16-1
      k ← 2 * i + 1
      WR[wd]8i+7+WRLEN/2..8i+WRLEN/2 ← WR[ws]8k+7..8k
      WR[wd]8i+7..8i ← WR[wt]8k+7..8k
   endfor

PCKOD.H
   for i in 0 .. WRLEN/32-1
      k ← 2 * i + 1
      WR[wd]16i+15+WRLEN/2..16i+WRLEN/2 ← WR[ws]16k+15..16k
      WR[wd]16i+15..16i ← WR[wt]16k+15..16k
   endfor

PCKOD.W
   for i in 0 .. WRLEN/64-1
      k ← 2 * i + 1
      WR[wd]32i+31+WRLEN/2..32i+WRLEN/2 ← WR[ws]32k+31..32k
      WR[wd]32i+31..32i ← WR[wt]32k+31..32k
   endfor

PCKOD.D
   for i in 0 .. WRLEN/128-1
      k ← 2 * i + 1
      WR[wd]64i+63+WRLEN/2..64i+WRLEN/2 ← WR[ws]64k+63..64k
      WR[wd]64i+63..64i ← WR[wt]64k+63..64k
   endfor
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception.




**Format:** `PCNT.df`

`PCNT.B wd,ws` **MSA**

`PCNT.H wd,ws` **MSA**

`PCNT.W wd,ws` **MSA**

`PCNT.D wd,ws` **MSA**

**Purpose:** Vector Population Count

Vector element count of all bits set to 1.

**Description:** `wd[i] ← population_count(ws[i])`

The number of bits set to 1 for elements in vector *ws* is stored to the elements in vector *wd*.

The operands and results are values in integer data format *df*.

**Restrictions:**

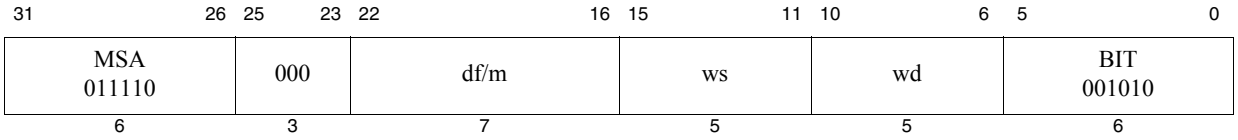
No data-dependent exceptions are possible.

**Operation:**

```
PCNT.B
   for i in 0 .. WRLEN/8-1
      WR[wd]8i+7..8i ← population_count(WR[ws]8i+7..8i, 8)
   endfor

PCNT.H
   for i in 0 .. WRLEN/16-1
      WR[wd]16i+15..16i ← population_count(WR[ws]16i+15..16i, 16)
   endfor

PCNT.W
   for i in 0 .. WRLEN/32-1
      WR[wd]32i+31..32i ← population_count(WR[ws]32i+31..32i, 32)
   endfor

PCNT.D
   for i in 0 .. WRLEN/64-1
      WR[wd]64i+63..64i ← population_count(WR[ws]64i+63..64i, 64)
   endfor

function population_count(tt, n)
   z ← 0
   for i in n-1..0
      if tti = 1 then
         z ← z + 1
      endif
endfunction population_count
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception.




**Format:** `SAT_S.df`

`SAT_S.B wd,ws,m` **MSA**

`SAT_S.H wd,ws,m` **MSA**

`SAT_S.W wd,ws,m` **MSA**

`SAT_S.D wd,ws,m` **MSA**

**Purpose:** Immediate Signed Saturate

Immediate selected bit width saturation of signed values.

**Description:** `wd[i] ← saturate_signed(ws[i], m+1)`

Signed elements in vector *ws* are saturated to signed values of *m+1* bits without changing the data width. The result is written to vector *wd*.

The operands and results are values in integer data format *df*.

**Restrictions:**

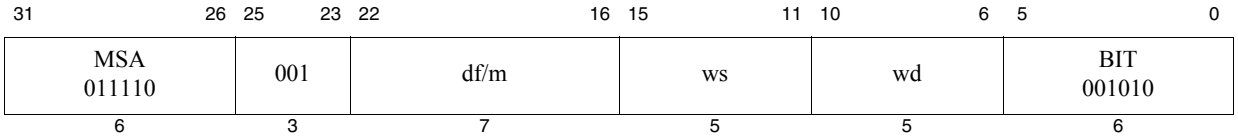
No data-dependent exceptions are possible.

**Operation:**

```
SAT_S.B
   for i in 0 .. WRLEN/8-1
      WR[wd]8i+7..8i ← sat_s(WR[ws]8i+7..8i, 8, m+1)
   endfor

SAT_S.H
   for i in 0 .. WRLEN/16-1
      WR[wd]16i+15..16i ← sat_s(WR[ws]16i+15..16i, 16, m+1)
   endfor

SAT_S.W
   for i in 0 .. WRLEN/32-1
      WR[wd]32i+31..32i ← sat_s(WR[ws]32i+31..32i, 32, m+1)
   endfor

SAT_S.D
   for i in 0 .. WRLEN/64-1
      WR[wd]64i+63..64i ← sat_s(WR[ws]64i+63..64i, 64, m+1)
   endfor

function sat_s(tt, n, b)
   if ttn-1 = 0 and ttn-1..b-1 ≠ 0n-b+1 then
      return 0n-b+1 || 1b-1
   endif
   if ttn-1 = 1 and ttn-1..b-1 ≠ 1n-b+1 then
      return 1n-b+1 || 0b-1
   else
      return tt
   endif
endfunction sat_s
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception.




**Format:** `SAT_U.df`

`SAT_U.B wd,ws,m` **MSA**

`SAT_U.H wd,ws,m` **MSA**

`SAT_U.W wd,ws,m` **MSA**

`SAT_U.D wd,ws,m` **MSA**

**Purpose:** Immediate Unsigned Saturate

Immediate selected bit width saturation of unsigned values.

**Description:** `wd[i] ← saturate_unsigned(ws[i], m+1)`

Unsigned elements in vector *ws* are saturated to unsigned values of *m+1* bits without changing the data width. The result is written to vector *wd*.

The operands and results are values in integer data format *df*.

**Restrictions:**

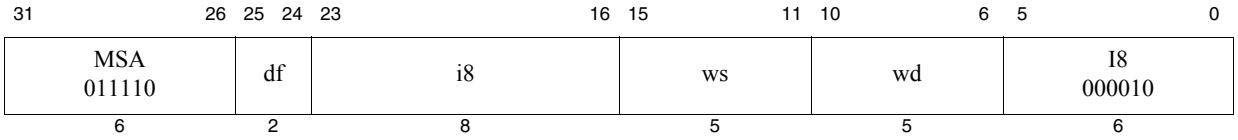
No data-dependent exceptions are possible.

**Operation:**

```
SAT_U.B
   for i in 0 .. WRLEN/8-1
      WR[wd]8i+7..8i ← sat_u(WR[ws]8i+7..8i, 8, m+1)
   endfor

SAT_U.H
   for i in 0 .. WRLEN/16-1
      WR[wd]16i+15..16i ← sat_u(WR[ws]16i+15..16i, 16, m+1)
   endfor

SAT_U.W
   for i in 0 .. WRLEN/32-1
      WR[wd]32i+31..32i ← sat_u(WR[ws]32i+31..32i, 32, m+1)
   endfor

SAT_U.D
   for i in 0 .. WRLEN/64-1
      WR[wd]64i+63..64i ← sat_u(WR[ws]64i+63..64i, 64, m+1)
   endfor

function sat_u(tt, n, b)
   if ttn-1..b ≠ 0n-b then
      return 0n-b || 1b
   else
      return tt
   endif
endfunction sat_u
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception.




**Format:** `SHF.df`

`SHF.B wd,ws,i8` **MSA**

`SHF.H wd,ws,i8` **MSA**

`SHF.W wd,ws,i8` **MSA**

**Purpose:** Immediate Set Shuffle Elements

Immediate control value-based 4 element set copy

**Description:** `wd[i] ← shuffle_set(ws, i, i8)`

The set shuffle instruction works on 4-element sets in *df* data format. All sets are shuffled in the same way: the element $i8_{2i+1..2i}$ in *ws* is copied over the element i in *wd*, where i is 0, 1, 2, 3.

The operands and results are values in byte data format.

**Restrictions:**

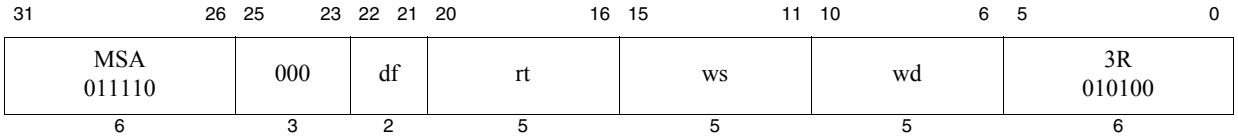
No data-dependent exceptions are possible.

**Operation:**

```
SHF.B
    for i in 0 .. WRLEN/8-1
        j ← i % 4
        k ← i - j + i8 2j+1..2j
        WR[wd] 8i+7..8i ← WR[ws] 8k+7..8k
    endfor

SHF.H
    for i in 0 .. WRLEN/16-1
        j ← i % 4
        k ← i - j + i8 2j+1..2j
        WR[wd] 16i+15..16i ← WR[ws] 16k+15..16k
    endfor

SHF.W
    for i in 0 .. WRLEN/32-1
        j ← i % 4
        k ← i - j + i8 2j+1..2j
        WR[wd] 32i+31..32i ← WR[ws] 32k+31..32k
    endfor
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception.




**Format:** `SLD.df`

`SLD.B wd,ws[rt]` **MSA**

`SLD.H wd,ws[rt]` **MSA**

`SLD.W wd,ws[rt]` **MSA**

`SLD.D wd,ws[rt]` **MSA**

**Purpose:** GPR Columns Slide

GPR number of columns to slide left source array.

**Description:** `wd[i] ← slide(wd, ws, rt)`

Vector registers *wd* and *ws* contain 2-dimensional byte arrays (rectangles) stored row-wise, with as many rows as bytes in integer data format *df*.

The slide instructions manipulate the content of vector registers *wd* and *ws* as byte elements, with data format *df* indicating the 2-dimensional byte array layout.

The two source rectangles *wd* and *ws* are concatenated horizontally in the order they appear in the syntax, i.e. first *wd* and then *ws*. Place a new destination rectangle over *ws* and then slide it to the left over the concatenation of *wd* and *ws* by the number of columns given in GPR *rt*. The result is written to vector *wd*.

GPR *rt* value is interpreted modulo the number of columns in destination rectangle, or equivalently, the number of data format *df* elements in the destination vector.

**Restrictions:**

No data-dependent exceptions are possible.

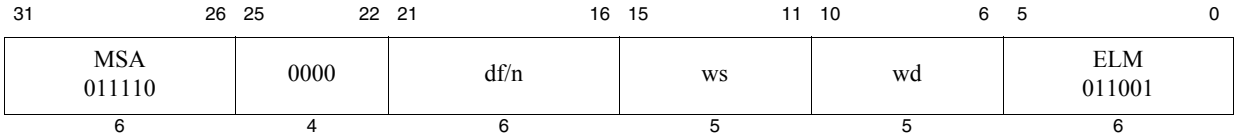
**Operation:**

```
SLD.B
    n ← GPR[rt] % (WRLEN/8)
    v ← WR[wd] || WR[ws]
    for i in 0 .. WRLEN/8-1
        j ← i + n
        WR[wd]8i+7..8i ← v8j+7..8j
    endfor

SLD.H
    n ← GPR[rt] % (WRLEN/16)
    s ← WRLEN/2
    for k in 0, 1
        t = s * k
        v ← (WR[wd]t+s-1..t || WR[ws]t+s-1..t)
        for i in 0 .. s/8-1
            j ← i + n
            WR[wd]t+8i+7..t+8i ← v8j+7..8j
        endfor
    endfor

SLD.W
    n ← GPR[rt] % (WRLEN/32)
    s ← WRLEN/4
```

```
        for k in 0, .., 3
            t = s * k
            v ← (WR[wd]t+s-1..t || WR[ws]t+s-1..t)
            for i in 0 .. s/8-1
                j ← i + n
                WR[wd]t+8i+7..t+8i ← v8j+7..8j
            endfor
        endfor

    SLD.D
        n ← GPR[rt] % (WRLEN/64)
        s ← WRLEN/8
        for k in 0, .., 7
            t = s * k
            v ← (WR[wd]t+s-1..t || WR[ws]t+s-1..t)
            for i in 0 .. s/8-1
                j ← i + n
                WR[wd]t+8i+7..t+8i ← v8j+7..8j
            endfor
        endfor
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception.




**Format:** `SLDI.df`

`SLDI.B wd,ws[n]` **MSA**

`SLDI.H wd,ws[n]` **MSA**

`SLDI.W wd,ws[n]` **MSA**

`SLDI.D wd,ws[n]` **MSA**

**Purpose:** Immediate Columns Slide

Immediate number of columns to slide left source array.

**Description:** `wd[i] ← slide(wd, ws, n)`

Vector registers *wd* and *ws* contain 2-dimensional byte arrays (rectangles) stored row-wise, with as many rows as bytes in integer data format *df*.

The slide instructions manipulate the content of vector registers *wd* and *ws* as byte elements, with data format *df* indicating the 2-dimensional byte array layout.

The two source rectangles *wd* and *ws* are concatenated horizontally in the order they appear in the syntax, i.e. first *wd* and then *ws*. Place a new destination rectangle over *ws* and then slide it to the left over the concatenation of *wd* and *ws* by *n* columns. The result is written to vector *wd*.

**Restrictions:**

No data-dependent exceptions are possible.

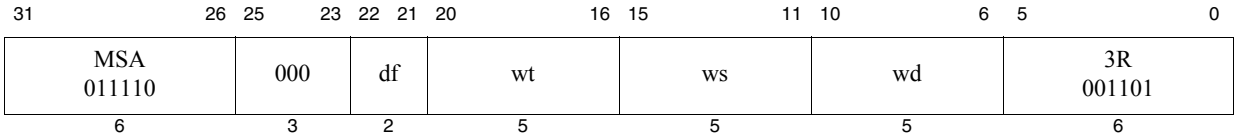
**Operation:**

```
SLDI.B
    v ← WR[wd] || WR[ws]
    for i in 0 .. WRLEN/8-1
        j ← i + n
        WR[wd]8i+7..8i ← v8j+7..8j
    endfor

SLDI.H
    s ← WRLEN/2
    for k in 0, 1
        t = s * k
        v ← (WR[wd]t+s-1..t || WR[ws]t+s-1..t)
        for i in 0 .. s/8-1
            j ← i + n
            WR[wd]t+8i+7..t+8i ← v8j+7..8j
        endfor
    endfor

SLDI.W
    s ← WRLEN/4
    for k in 0, .., 3
        t = s * k
        v ← (WR[wd]t+s-1..t || WR[ws]t+s-1..t)
        for i in 0 .. s/8-1
            j ← i + n
```

```
            WR[wd]t+8i+7..t+8i ← v8j+7..8j
        endfor
    endfor

SLDI.D
    s ← WRLEN/8
    for k in 0, .., 7
        t = s * k
        v ← (WR[wd]t+s-1..t || WR[ws]t+s-1..t)
        for i in 0 .. s/8-1
            j ← i + n
            WR[wd]t+8i+7..t+8i ← v8j+7..8j
        endfor
    endfor
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception.




**Format:** `SLL.df`

`SLL.B wd,ws,wt` **MSA**

`SLL.H wd,ws,wt` **MSA**

`SLL.W wd,ws,wt` **MSA**

`SLL.D wd,ws,wt` **MSA**

**Purpose:** Vector Shift Left

Vector bit count shift left.

**Description:** `wd[i] ← ws[i] << wt[i]`

The elements in vector *ws* are shifted left by the number of bits the elements in vector *wt* specify modulo the size of the element in bits. The result is written to vector *wd*.

The operands and results are values in integer data format *df*.

**Restrictions:**

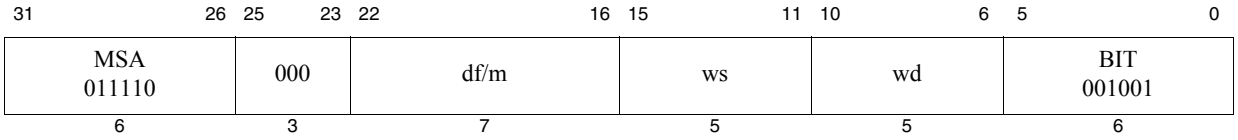
No data-dependent exceptions are possible.

**Operation:**

```
SLL.B
    for i in 0 .. WRLEN/8-1
        t ← WR[wt]8i+2..8i
        WR[wd]8i+7..8i ← WR[ws]8i+8-t-1..8i || 0t
    endfor

SLL.H
    for i in 0 .. WRLEN/16-1
        t ← WR[wt]16i+3..16i
        WR[wd]16i+15..16i ← WR[ws]16i+16-t-1..16i || 0t
    endfor

SLL.W
    for i in 0 .. WRLEN/32-1
        t ← WR[wt]32i+4..32i
        WR[wd]32i+31..32i ← WR[ws]32i+32-t-1..32i || 0t
    endfor

SLL.D
    for i in 0 .. WRLEN/64-1
        t ← WR[wt]64i+5..64i
        WR[wd]64i+63..64i ← WR[ws]64i+64-t-1..64i || 0t
    endfor
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception.




**Format:** `SLLI.df`

`SLLI.B wd,ws,m` **MSA**

`SLLI.H wd,ws,m` **MSA**

`SLLI.W wd,ws,m` **MSA**

`SLLI.D wd,ws,m` **MSA**

**Purpose:** Immediate Shift Left

Immediate bit count shift left.

**Description:** `wd[i] ← ws[i] << m`

The elements in vector *ws* are shifted left by *m* bits. The result is written to vector *wd*.

The operands and results are values in integer data format *df*.

**Restrictions:**

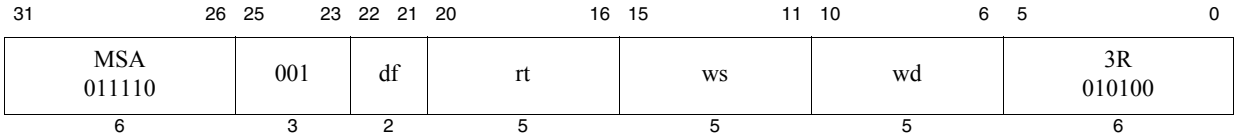
No data-dependent exceptions are possible.

**Operation:**

```
SLLI.B
    t ← m
    for i in 0 .. WRLEN/8-1
        WR[wd]8i+7..8i ← WR[ws]8i+8-t-1..8i || 0^t
    endfor

SLLI.H
    t ← m
    for i in 0 .. WRLEN/16-1
        WR[wd]16i+15..16i ← WR[ws]16i+16-t-1..16i || 0^t
    endfor

SLLI.W
    t ← m
    for i in 0 .. WRLEN/32-1
        WR[wd]32i+31..32i ← WR[ws]32i+32-t-1..32i || 0^t
    endfor

SLLI.D
    t ← m
    for i in 0 .. WRLEN/64-1
        WR[wd]64i+63..64i ← WR[ws]64i+64-t-1..64i || 0^t
    endfor
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception.




**Format:** `SPLAT.df`

`SPLAT.B wd,ws[rt]` **MSA**

`SPLAT.H wd,ws[rt]` **MSA**

`SPLAT.W wd,ws[rt]` **MSA**

`SPLAT.D wd,ws[rt]` **MSA**

**Purpose:** GPR Element Splat

GPR selected element replicated in all destination elements.

**Description:** `wd[i] ← ws[rt]`

Replicate vector *ws* element with index given by GPR *rt* to all elements in vector *wd*.

GPR *rt* value is interpreted modulo the number of data format *df* elements in the destination vector.

The operands and results are values in data format *df*.

**Restrictions:**

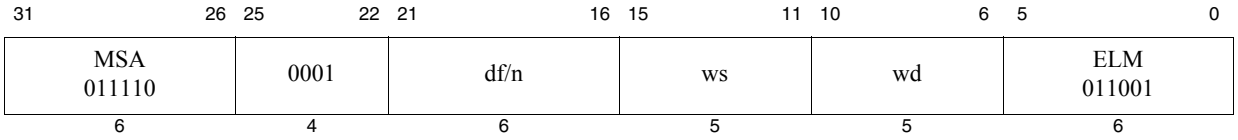
No data-dependent exceptions are possible.

**Operation:**

```
SPLAT.B
    n ← GPR[rt] % (WRLEN/8)
    for i in 0 .. WRLEN/8-1
        WR[wd]8i+7..8i ← WR[ws]8n+7..8n
    endfor

SPLAT.H
    n ← GPR[rt] % (WRLEN/16)
    for i in 0 .. WRLEN/16-1
        WR[wd]16i+15..16i ← WR[ws]16n+15..16n
    endfor

SPLAT.W
    n ← GPR[rt] % (WRLEN/32)
    for i in 0 .. WRLEN/32-1
        WR[wd]32i+31..32i ← WR[ws]32n+31..32n
    endfor

SPLAT.D
    n ← GPR[rt] % (WRLEN/64)
    for i in 0 .. WRLEN/64-1
        WR[wd]64i+63..64i ← WR[ws]64n+63..64n
    endfor
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception.




**Format:** `SPLATI.df`

`SPLATI.B wd,ws[n]` **MSA**

`SPLATI.H wd,ws[n]` **MSA**

`SPLATI.W wd,ws[n]` **MSA**

`SPLATI.D wd,ws[n]` **MSA**

**Purpose:** Immediate Element Splat

Immediate selected element replicated in all destination elements.

**Description:** `wd[i] ← ws[n]`

Replicate element *n* in vector *ws* to all elements in vector *wd*.

The operands and results are values in data format *df*.

**Restrictions:**

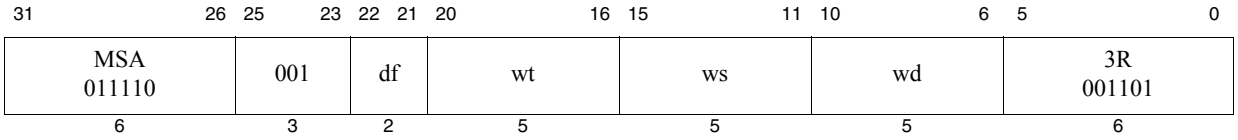
No data-dependent exceptions are possible.

**Operation:**

```
SPLATI.B
    for i in 0 .. WRLEN/8-1
        WR[wd]8i+7..8i ← WR[ws]8n+7..8n
    endfor

SPLATI.H
    for i in 0 .. WRLEN/16-1
        WR[wd]16i+15..16i ← WR[ws]16n+15..16n
    endfor

SPLATI.W
    for i in 0 .. WRLEN/32-1
        WR[wd]32i+31..32i ← WR[ws]32n+31..32n
    endfor

SPLATI.D
    for i in 0 .. WRLEN/64-1
        WR[wd]64i+63..64i ← WR[ws]64n+63..64n
    endfor
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception.




**Format:** `SRA.df`

`SRA.B wd,ws,wt` **MSA**

`SRA.H wd,ws,wt` **MSA**

`SRA.W wd,ws,wt` **MSA**

`SRA.D wd,ws,wt` **MSA**

**Purpose:** Vector Shift Right Arithmetic

Vector bit count shift right arithmetic.

**Description:** `wd[i] ← ws[i] >> wt[i]`

The elements in vector *ws* are shifted right arithmetic by the number of bits the elements in vector *wt* specify modulo the size of the element in bits. The result is written to vector *wd*.

The operands and results are values in integer data format *df*.

**Restrictions:**

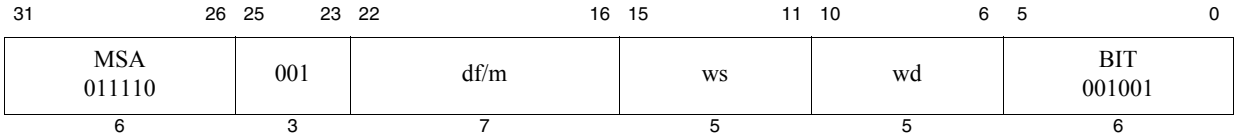
No data-dependent exceptions are possible.

**Operation:**

```
SRA.B
    for i in 0 .. WRLEN/8-1
        t ← WR[wt]8i+2..8i
        WR[wd]8i+7..8i ← (WR[ws]8i+7)t || WR[ws]8i+7..8i+t
    endfor

SRA.H
    for i in 0 .. WRLEN/16-1
        t ← WR[wt]16i+3..16i
        WR[wd]16i+15..16i ← (WR[ws]16i+15)t || WR[ws]16i+15..16i+t
    endfor

SRA.W
    for i in 0 .. WRLEN/32-1
        t ← WR[wt]32i+4..32i
        WR[wd]32i+31..32i ← (WR[ws]32i+31)t || WR[ws]32i+31..32i+t
    endfor

SRA.D
    for i in 0 .. WRLEN/64-1
        t ← WR[wt]64i+5..64i
        WR[wd]64i+63..64i ← (WR[ws]64i+63)t || WR[ws]64i+63..64i+t
    endfor
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception.




**Format:** `SRAI.df`

`SRAI.B wd,ws,m` **MSA**

`SRAI.H wd,ws,m` **MSA**

`SRAI.W wd,ws,m` **MSA**

`SRAI.D wd,ws,m` **MSA**

**Purpose:** Immediate Shift Right Arithmetic

Immediate bit count shift right arithmetic.

**Description:** `wd[i] ← ws[i] >> m`

The elements in vector *ws* are shifted right arithmetic by *m* bits. The result is written to vector *wd*.

The operands and results are values in integer data format *df*.

**Restrictions:**

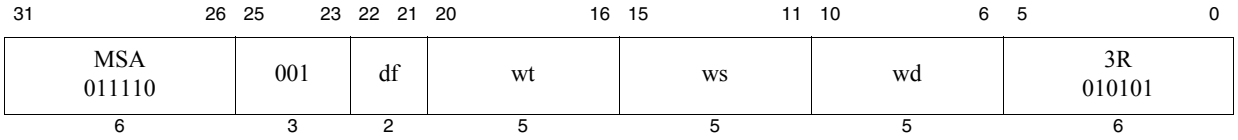
No data-dependent exceptions are possible.

**Operation:**

```
SRAI.B
    t ← m
    for i in 0 .. WRLEN/8-1
        WR[wd]8i+7..8i ← (WR[ws]8i+7)t || WR[ws]8i+7..8i+t
    endfor

SRAI.H
    t ← m
    for i in 0 .. WRLEN/16-1
        WR[wd]16i+15..16i ← (WR[ws]16i+15)t || WR[ws]16i+15..16i+t
    endfor

SRAI.W
    t ← m
    for i in 0 .. WRLEN/32-1
        WR[wd]32i+31..32i ← (WR[ws]32i+31)t || WR[ws]32i+31..32i+t
    endfor

SRAI.D
    t ← m
    for i in 0 .. WRLEN/64-1
        WR[wd]64i+63..64i ← (WR[ws]64i+63)t || WR[ws]64i+63..64i+t
    endfor
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception.




**Format:** `SRAR.df`

`SRAR.B wd,ws,wt` **MSA**

`SRAR.H wd,ws,wt` **MSA**

`SRAR.W wd,ws,wt` **MSA**

`SRAR.D wd,ws,wt` **MSA**

**Purpose:** Vector Shift Right Arithmetic Rounded

Vector bit count shift right arithmetic with rounding

**Description:** `wd[i] ← ws[i] >>(rounded) wt[i]`

The elements in vector *ws* are shifted right arithmetic by the number of bits the elements in vector *wt* specify modulo the size of the element in bits. The most significant discarded bit is added to the shifted value (for rounding) and the result is written to vector *wd*.

The operands and results are values in integer data format *df*.

**Restrictions:**

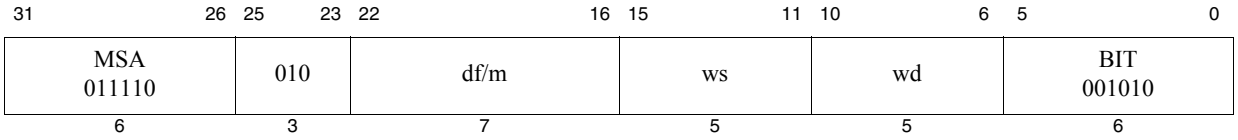
No data-dependent exceptions are possible.

**Operation:**

```
SRAR.B
   for i in 0 .. WRLEN/8-1
      WR[wd]8i+7..8i ← srar(WR[ws]8i+7..8i,  WR[wt]8i+2..8i, 8)
   endfor

SRAR.H
   for i in 0 .. WRLEN/16-1
      WR[wd]16i+15..16i ← srar(WR[ws]16i+15..16i, WR[wt]16i+3..16i, 16)
   endfor

SRAR.W
   for i in 0 .. WRLEN/32-1
      WR[wd]32i+31..32i ← srar(WR[ws]32i+31..32i, WR[wt]32i+4..32i, 32)
   endfor

SRAR.D
   for i in 0 .. WRLEN/64-1
      WR[wd]64i+63..64i ← srar(WR[ws]64i+63..64i, WR[wt]64i+5..64i, 64)
   endfor

function srar(ts, n, b)
   if n = 0 then
      return ts
   else
      return ((tsb-1)n || tsb-1..n) + tsn-1
   endif
endfunction srar
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception.




**Format:** `SRARI.df`

`SRARI.B wd,ws,m` **MSA**

`SRARI.H wd,ws,m` **MSA**

`SRARI.W wd,ws,m` **MSA**

`SRARI.D wd,ws,m` **MSA**

**Purpose:** Immediate Shift Right Arithmetic Rounded

Immediate bit count shift right arithmetic with rounding

**Description:** `wd[i] ← ws[i] >>(rounded) m`

The elements in vector *ws* are shifted right arithmetic by *m* bits. The most significant discarded bit is added to the shifted value (for rounding) and the result is written to vector *wd*.

The operands and results are values in integer data format *df*.

**Restrictions:**

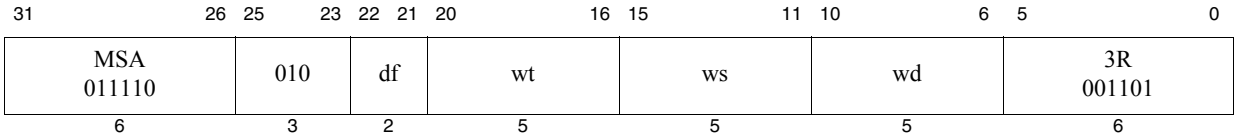
No data-dependent exceptions are possible.

**Operation:**

```
SRARI.B
   for i in 0 .. WRLEN/8-1
      WR[wd]8i+7..8i ← srar(WR[ws]8i+7..8i,  m, 8)
   endfor

SRARI.H
   for i in 0 .. WRLEN/16-1
      WR[wd]16i+15..16i ← srar(WR[ws]16i+15..16i, m, 16)
   endfor

SRARI.W
   for i in 0 .. WRLEN/32-1
      WR[wd]32i+31..32i ← srar(WR[ws]32i+31..32i, m, 32)
   endfor

SRARI.D
   for i in 0 .. WRLEN/64-1
      WR[wd]64i+63..64i ← srar(WR[ws]64i+63..64i, m, 64)
   endfor

function srar(ts, n, b)
   if n = 0 then
      return ts
   else
      return ((tsb-1)n || tsb-1..n) + tsn-1
   endif
endfunction srar
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception.




**Format:** `SRL.df`

`SRL.B wd,ws,wt` **MSA**

`SRL.H wd,ws,wt` **MSA**

`SRL.W wd,ws,wt` **MSA**

`SRL.D wd,ws,wt` **MSA**

**Purpose:** Vector Shift Right Logical

Vector bit count shift right logical.

**Description:** `wd[i] ← ws[i] >> wt[i]`

The elements in vector *ws* are shifted right logical by the number of bits the elements in vector *wt* specify modulo the size of the element in bits. The result is written to vector *wd*.

The operands and results are values in integer data format *df*.

**Restrictions:**

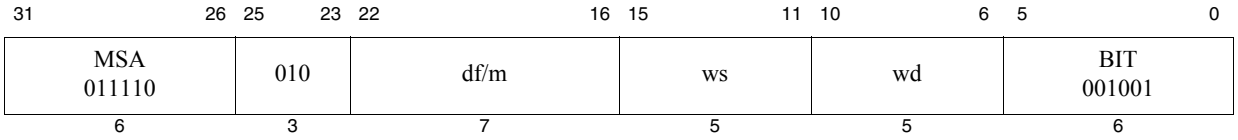
No data-dependent exceptions are possible.

**Operation:**

```
SRL.B
    for i in 0 .. WRLEN/8-1
        t ← WR[wt]8i+2..8i
        WR[wd]8i+7..8i ← 0^t || WR[ws]8i+7..8i+t
    endfor

SRL.H
    for i in 0 .. WRLEN/16-1
        t ← WR[wt]16i+3..16i
        WR[wd]16i+15..16i ← 0^t || WR[ws]16i+15..16i+t
    endfor

SRL.W
    for i in 0 .. WRLEN/32-1
        t ← WR[wt]32i+4..32i
        WR[wd]32i+31..32i ← 0^t || WR[ws]32i+31..32i+t
    endfor

SRL.D
    for i in 0 .. WRLEN/64-1
        t ← WR[wt]64i+5..64i
        WR[wd]64i+63..64i ← (WR[ws]64i+63)^t || WR[ws]64i+63..64i+t
    endfor
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception.




**Format:** `SRLI.df`

`SRLI.B wd,ws,m` **MSA**

`SRLI.H wd,ws,m` **MSA**

`SRLI.W wd,ws,m` **MSA**

`SRLI.D wd,ws,m` **MSA**

**Purpose:** Immediate Shift Right Logical

Immediate bit count shift right logical.

**Description:** `wd[i] ← ws[i] >> m`

The elements in vector *ws* are shifted right logical by *m* bits. The result is written to vector *wd*.

The operands and results are values in integer data format *df*.

**Restrictions:**

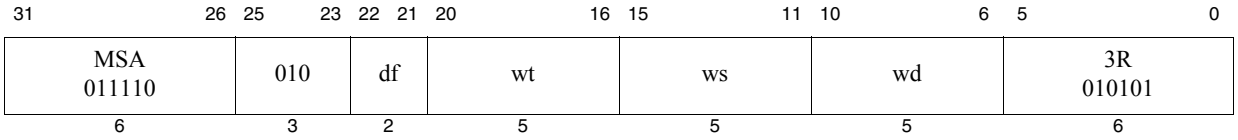
No data-dependent exceptions are possible.

**Operation:**

```
SRLI.B
    t ← m
    for i in 0 .. WRLEN/8-1
        WR[wd]8i+7..8i ← 0^t || WR[ws]8i+7..8i+t
    endfor

SRLI.H
    t ← m
    for i in 0 .. WRLEN/16-1
        WR[wd]16i+15..16i ← 0^t || WR[ws]16i+15..16i+t
    endfor

SRLI.W
    t ← m
    for i in 0 .. WRLEN/32-1
        WR[wd]32i+31..32i ← 0^t || WR[ws]32i+31..32i+t
    endfor

SRLI.D
    t ← m
    for i in 0 .. WRLEN/64-1
        WR[wd]64i+63..64i ← 0^t || WR[ws]64i+63..64i+t
    endfor
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception.




**Format:** `SRLR.df`

`SRLR.B wd,ws,wt` **MSA**

`SRLR.H wd,ws,wt` **MSA**

`SRLR.W wd,ws,wt` **MSA**

`SRLR.D wd,ws,wt` **MSA**

**Purpose:** Vector Shift Right Logical Rounded

Vector bit count shift right logical with rounding

**Description:** `wd[i] ← ws[i] >>(rounded) wt[i]`

The elements in vector *ws* are shifted right logical by the number of bits the elements in vector *wt* specify modulo the size of the element in bits. The most significant discarded bit is added to the shifted value (for rounding) and the result is written to vector *wd*.

The operands and results are values in integer data format *df*.

**Restrictions:**

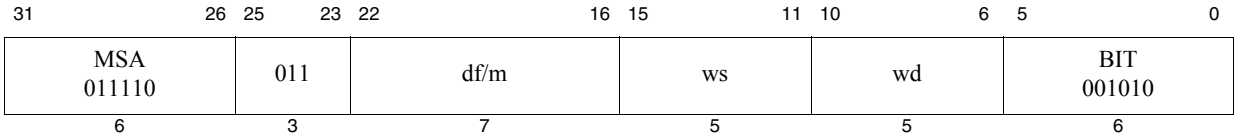
No data-dependent exceptions are possible.

**Operation:**

```
SRLR.B
    for i in 0 .. WRLEN/8-1
        WR[wd]8i+7..8i ← srlr(WR[ws]8i+7..8i, WR[wt]8i+2..8i, 8)
    endfor

SRLR.H
    for i in 0 .. WRLEN/16-1
        WR[wd]16i+15..16i ← srlr(WR[ws]16i+15..16i, WR[wt]16i+3..16i, 16)
    endfor

SRLR.W
    for i in 0 .. WRLEN/32-1
        WR[wd]32i+31..32i ← srlr(WR[ws]32i+31..32i, WR[wt]32i+4..32i, 32)
    endfor

SRLR.D
    for i in 0 .. WRLEN/64-1
        WR[wd]64i+63..64i ← srlr(WR[ws]64i+63..64i, WR[wt]64i+5..64i, 64)
    endfor

function srlr(ts, n, b)
    if n = 0 then
        return ts
    else
        return (0^n || ts_b-1..n) + ts_n-1
    endif
endfunction srlr
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception.




**Format:** `SRLRI.df`

`SRLRI.B wd,ws,m` **MSA**

`SRLRI.H wd,ws,m` **MSA**

`SRLRI.W wd,ws,m` **MSA**

`SRLRI.D wd,ws,m` **MSA**

**Purpose:** Immediate Shift Right Logical Rounded

Immediate bit count shift right logical with rounding

**Description:** `wd[i] ← ws[i] >>(rounded) m`

The elements in vector *ws* are shifted right logical by *m* bits. The most significant discarded bit is added to the shifted value (for rounding) and the result is written to vector *wd*.

The operands and results are values in integer data format *df*.

**Restrictions:**

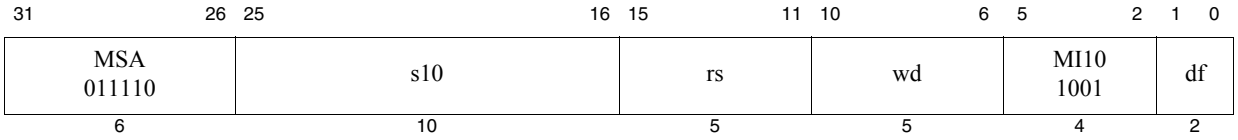
No data-dependent exceptions are possible.

**Operation:**

```
SRLRI.B
   for i in 0 .. WRLEN/8-1
      WR[wd]8i+7..8i ← srlr(WR[ws]8i+7..8i, m, 8)
   endfor

SRLRI.H
   for i in 0 .. WRLEN/16-1
      WR[wd]16i+15..16i ← srlr(WR[ws]16i+15..16i, m, 16)
   endfor

SRLRI.W
   for i in 0 .. WRLEN/32-1
      WR[wd]32i+31..32i ← srlr(WR[ws]32i+31..32i, m, 32)
   endfor

SRLRI.D
   for i in 0 .. WRLEN/64-1
      WR[wd]64i+63..64i ← srlr(WR[ws]64i+63..64i, m, 64)
   endfor

function srlr(ts, n, b)
   if n = 0 then
      return ts
   else
      return (0^n || tsb-1..n) + tsn-1
   endif
endfunction srlr
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception.




**Format:** `ST.df`

`ST.B wd,s10(rs)` **MSA**

`ST.H wd,s10(rs)` **MSA**

`ST.W wd,s10(rs)` **MSA**

`ST.D wd,s10(rs)` **MSA**

**Purpose:** Vector Store

Vector store element-by-element to base register plus offset memory address.

**Description:** `memory[rs + s10 + i * sizeof(wd[i])] ← wd[i]`

The *WRLEN* / 8 bytes in vector *wd* are stored as elements of data format *df* at the effective memory location addressed by the base *rs* and the 10-bit signed immediate offset *s10*.

The *s10* offset in data format *df* units is added to the base *rs* to form the effective memory location address. *rs* and the effective memory location address have no alignment restrictions.

If the effective memory location address is element aligned, the vector store instruction is atomic at the element level with no guaranteed ordering among elemen ts, i.e. each element store is an atomic operation issued in no particular order with respect to the element's vector position.

By convention, in the assembly language syntax all offsets are in bytes and have to be multiple of the size of the data format *df*. The assembler determines the *s10* bitfield value dividing the byte offset by the size of the data format *df*.

**Restrictions:**

Address-dependent exceptions are possible.

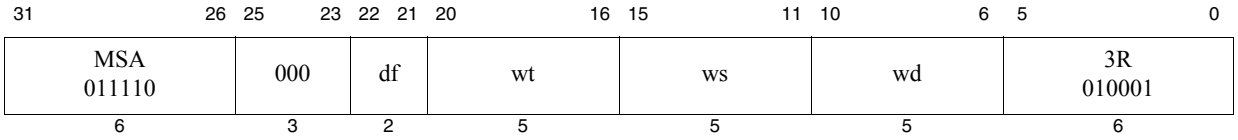
**Operation:**

```
ST.B
   a ← rs + s10
   StoreByteVector(WR[wd]WRLEN-1..0, a, WRLEN/8)

ST.H
   a ← rs + s10 * 2
   StoreHalfwordVector(WR[wd]WRLEN-1..0, a, WRLEN/16)

ST.W
   a ← rs + s10 * 4
   StoreWordVector(WR[wd]WRLEN-1..0, a, WRLEN/32)

ST.D
   a ← rs + s10 * 8
   StoreDoublewordVector(WR[wd]WRLEN-1..0, a, WRLEN/64)

function StoreByteVector(tt, a, n)
   /* Implementation defined store n byte vector tt to virtual
         address a. */
endfunction StoreByteVector

function StoreHalfwordVector(tt, a, n)
   /* Implementation defined store n halfword vector tt to virtual
```

```
          address a. */
endfunction StoreHalfwordVector

function StoreWordVector(tt, a, n)
   /* Implementation defined store n word vector tt to virtual
          address a. */
endfunction StoreWordVector

function StoreDoublewordVector(tt, a, n)
   /* Implementation defined store n doubleword vector tt to virtual
          address a. */
endfunction StoreDoublewordVector
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception. Data access TLB and Address Error Exceptions.




**Format:** `SUBS_S.df`

`SUBS_S.B wd,ws,wt` **MSA**

`SUBS_S.H wd,ws,wt` **MSA**

`SUBS_S.W wd,ws,wt` **MSA**

`SUBS_S.D wd,ws,wt` **MSA**

**Purpose:** Vector Signed Saturated Subtract of Signed Values

Vector subtraction from vector saturating the result as signed value.

**Description:** `wd[i] ← saturate_signed(signed(ws[i]) - signed(wt[i]))`

The elements in vector *wt* are subtracted from the elements in vector *ws*. Signed arithmetic is performed and overflows clamp to the largest and/or smallest representable signed values before writing the result to vector *wd*.

The operands and results are values in integer data format *df*.

**Restrictions:**

No data-dependent exceptions are possible.

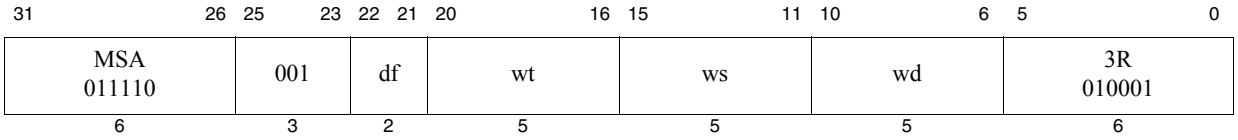
**Operation:**

```
SUBS_S.B
    for i in 0 .. WRLEN/8-1
        WR[wd]8i+7..8i ← subs_s(WR[ws]8i+7..8i, WR[wt]8i+7..8i, 8)
    endfor

SUBS_S.H
    for i in 0 .. WRLEN/16-1
        WR[wd]16i+15..16i ← subs_s(WR[ws]16i+15..16i, WR[wt]16i+15..16i, 16)
    endfor

SUBS_S.W
    for i in 0 .. WRLEN/32-1
        WR[wd]32i+31..32i ← subs_s(WR[ws]32i+31..32i, WR[wt]32i+31..32i, 32)
    endfor

SUBS_S.D
    for i in 0 .. WRLEN/64-1
        WR[wd]64i+63..64i ← subs_s(WR[ws]64i+63..64i, WR[wt]64i+63..64i, 64)
    endfor

function sat_s(tt, n, b)
    if ttn-1 = 0 and ttn-1..b-1 ≠ 0n-b+1 then
        return 0n-b+1 || 1b-1
    endif
    if ttn-1 = 1 and ttn-1..b-1 ≠ 1n-b+1 then
        return 1n-b+1 || 0b-1
    else
        return tt
    endif
endfunction sat_s
```

```
function subs_s(ts, tt, n)
   t ← (ts_{n-1} || ts) - (tt_{n-1} || tt)
   return sat_s(t, n+1, n)
endfunction subs_s
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception.




**Format:** `SUBS_U.df`

`SUBS_U.B wd,ws,wt` **MSA**

`SUBS_U.H wd,ws,wt` **MSA**

`SUBS_U.W wd,ws,wt` **MSA**

`SUBS_U.D wd,ws,wt` **MSA**

**Purpose:** Vector Unsigned Saturated Subtract of Unsigned Values

Vector subtraction from vector saturating the result as unsigned value.

**Description:** `wd[i] ← saturate_unsigned(unsigned(ws[i]) - unsigned(wt[i]))`

The elements in vector *wt* are subtracted from the elements in vector *ws*. Unsigned arithmetic is performed and underflows clamp to 0 before writing the result to vector *wd*.

The operands and results are values in integer data format *df*.

**Restrictions:**

No data-dependent exceptions are possible.

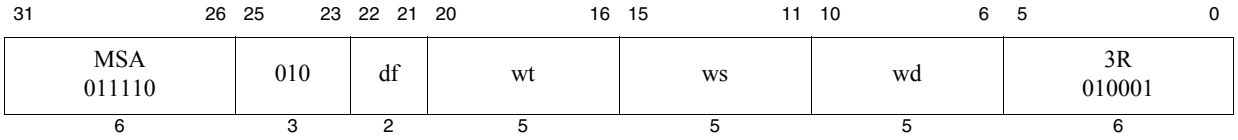
**Operation:**

```
SUBS_U.B
   for i in 0 .. WRLEN/8-1
      WR[wd]8i+7..8i ← subs_u(WR[ws]8i+7..8i, WR[wt]8i+7..8i, 8)
   endfor

SUBS_U.H
   for i in 0 .. WRLEN/16-1
      WR[wd]16i+15..16i ← subs_u(WR[ws]16i+15..16i, WR[wt]16i+15..16i, 16)
   endfor

SUBS_U.W
   for i in 0 .. WRLEN/32-1
      WR[wd]32i+31..32i ← subs_u(WR[ws]32i+31..32i, WR[wt]32i+31..32i, 32)
   endfor

SUBS_U.D
   for i in 0 .. WRLEN/64-1
      WR[wd]64i+63..64i ← subs_u(WR[ws]64i+63..64i, WR[wt]64i+63..64i, 64)
   endfor

function sat_u(tt, n, b)
   if ttn-1..b ≠ 0n-b then
      return 0n-b || 1b
   else
      return tt
   endif
endfunction sat_u

function subs_u(ts, tt, n)
   t ← (0 || ts) - (0 || tt)
```

```
        if t_n = 0
            return sat_u(t, n+1, n)
        else
            return 0
    endfunction subs_u
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception.




**Format:** `SUBSUS_U.df`

`SUBSUS_U.B wd,ws,wt` **MSA**

`SUBSUS_U.H wd,ws,wt` **MSA**

`SUBSUS_U.W wd,ws,wt` **MSA**

`SUBSUS_U.D wd,ws,wt` **MSA**

**Purpose:** Vector Unsigned Saturated Subtract of Signed from Unsigned

Vector subtraction of signed values from unsigned values saturating the results as unsigned values.

**Description:** `wd[i] ← saturate_unsigned(unsigned(ws[i]) - signed(wt[i]))`

The signed elements in vector *wt* are subtracted from the unsigned elements in vector *ws*. The signed result is unsigned saturated and written to vector *wd*.

The operands and results are values in integer data format *df*.

**Restrictions:**

No data-dependent exceptions are possible.

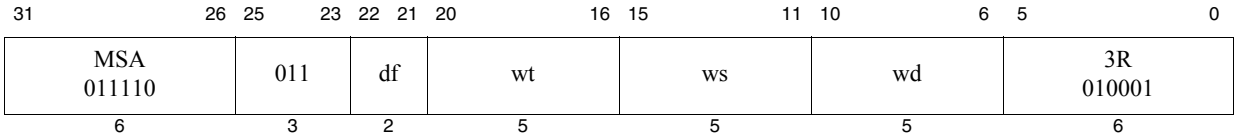
**Operation:**

```
SUBSUS_U.B
    for i in 0 .. WRLEN/8-1
        WR[wd]8i+7..8i ← subsus_u(WR[ws]8i+7..8i, WR[wt]8i+7..8i, 8)
    endfor

SUBSUS_U.H
    for i in 0 .. WRLEN/16-1
        WR[wd]16i+15..16i ← subsus_u(WR[ws]16i+15..16i, WR[wt]16i+15..16i, 16)
    endfor

SUBSUS_U.W
    for i in 0 .. WRLEN/32-1
        WR[wd]32i+31..32i ← subsus_u(WR[ws]32i+31..32i, WR[wt]32i+31..32i, 32)
    endfor

SUBSUS_U.D
    for i in 0 .. WRLEN/64-1
        WR[wd]64i+63..64i ← subsus_u(WR[ws]64i+63..64i, WR[wt]64i+63..64i, 64)
    endfor

function sat_u(tt, n, b)
    if ttn-1..b ≠ 0n-b then
        return 0n-b || 1b
    else
        return tt
    endif
endfunction sat_u

function subsus_u(ts, tt, n)
    t ← (0 || ts) - (ttn-1 || tt)
```

```
        if t_n = 0
            return sat_u(t, n+1, n)
        else
            return 0
    endfunction subsus_u
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception.




**Format:** `SUBSUU_S.df`

`SUBSUU_S.B wd,ws,wt` **MSA**

`SUBSUU_S.H wd,ws,wt` **MSA**

`SUBSUU_S.W wd,ws,wt` **MSA**

`SUBSUU_S.D wd,ws,wt` **MSA**

**Purpose:** Vector Signed Saturated Subtract of Unsigned Values

Vector subtraction from vector of unsigned values saturating the results as signed values.

**Description:** `wd[i] ← saturate_signed(unsigned(ws[i]) - unsigned(wt[i]))`

The unsigned elements in vector *wt* are subtracted from the unsigned elements in vector *ws*. The signed result is signed saturated and written to vector *wd*.

The operands and results are values in integer data format *df*.

**Restrictions:**

No data-dependent exceptions are possible.

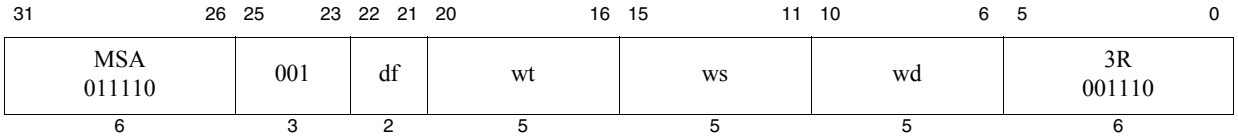
**Operation:**

```
SUBSUU_S.B
    for i in 0 .. WRLEN/8-1
        WR[wd]8i+7..8i ← subsuu_s(WR[ws]8i+7..8i, WR[wt]8i+7..8i, 8)
    endfor

SUBSUU_S.H
    for i in 0 .. WRLEN/16-1
        WR[wd]16i+15..16i ← subsuu_s(WR[ws]16i+15..16i, WR[wt]16i+15..16i, 16)
    endfor

SUBSUU_S.W
    for i in 0 .. WRLEN/32-1
        WR[wd]32i+31..32i ← subsuu_s(WR[ws]32i+31..32i, WR[wt]32i+31..32i, 32)
    endfor

SUBSUU_S.D
    for i in 0 .. WRLEN/64-1
        WR[wd]64i+63..64i ← subsuu_s(WR[ws]64i+63..64i, WR[wt]64i+63..64i, 64)
    endfor

function sat_s(tt, n, b)
    if ttn-1 = 0 and ttn-1..b-1 ≠ 0n-b+1 then
        return 0n-b+1 || 1b-1
    endif
    if ttn-1 = 1 and ttn-1..b-1 ≠ 1n-b+1 then
        return 1n-b+1 || 0b-1
    else
        return tt
    endif
endfunction sat_s
```

```
function subsuu_s(ts, tt, n)
   t ← (0 || ts) - (0 || tt)
   return sat_s(t, n+1, n)
endfunction subsuu_s
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception.




**Format:** `SUBV.df`

`SUBV.B wd,ws,wt` **MSA**

`SUBV.H wd,ws,wt` **MSA**

`SUBV.W wd,ws,wt` **MSA**

`SUBV.D wd,ws,wt` **MSA**

**Purpose:** Vector Subtract

Vector subtraction from vector.

**Description:** `wd[i] ← ws[i] - wt[i]`

The elements in vector *wt* are subtracted from the elements in vector *ws*. The result is written to vector *wd*.

The operands and results are values in integer data format *df*.

**Restrictions:**

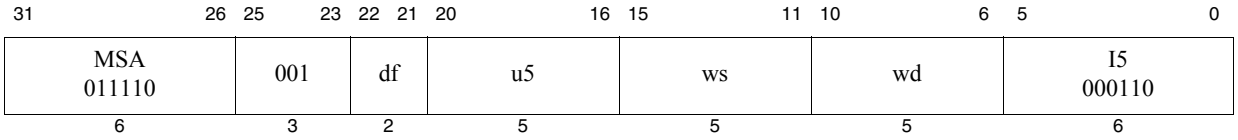
No data-dependent exceptions are possible.

**Operation:**

```
SUBV.B
    for i in 0 .. WRLEN/8-1
        WR[wd]8i+7..8i ← WR[ws]8i+7..8i - WR[wt]8i+7..8i
    endfor

SUBV.H
    for i in 0 .. WRLEN/16-1
        WR[wd]16i+15..16i ← WR[ws]16i+15..16i - WR[wt]16i+15..16i
    endfor

SUBV.W
    for i in 0 .. WRLEN/32-1
        WR[wd]32i+31..32i ← WR[ws]32i+31..32i - WR[wt]32i+31..32i
    endfor

SUBV.D
    for i in 0 .. WRLEN/64-1
        WR[wd]64i+63..64i ← WR[ws]64i+63..64i - WR[wt]64i+63..64i
    endfor
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception.




**Format:** `SUBVI.df`

`SUBVI.B wd,ws,u5` **MSA**

`SUBVI.H wd,ws,u5` **MSA**

`SUBVI.W wd,ws,u5` **MSA**

`SUBVI.D wd,ws,u5` **MSA**

**Purpose:** Immediate Subtract

Immediate subtraction from vector.

**Description:** `wd[i] ← ws[i] - u5`

The 5-bit immediate unsigned value *u5* is subtracted from the elements in vector *ws*. The result is written to vector *wd*.

The operands and results are values in integer data format *df*.

**Restrictions:**

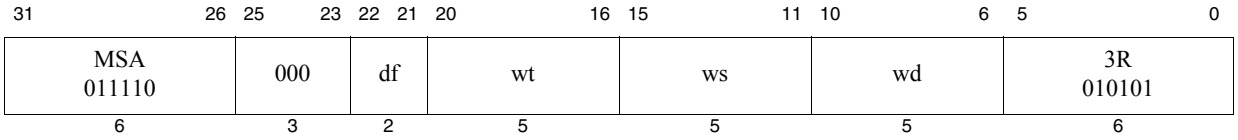
No data-dependent exceptions are possible.

**Operation:**

```
SUBVI.B
    t ← 0^3 || u5_4..0
    for i in 0 .. WRLEN/8-1
        WR[wd]_8i+7..8i ← WR[ws]_8i+7..8i - t
    endfor

SUBVI.H
    t ← 0^11 || u5_4..0
    for i in 0 .. WRLEN/16-1
        WR[wd]_16i+15..16i ← WR[ws]_16i+15..16i - t
    endfor

SUBVI.W
    t ← 0^27 || u5_4..0
    for i in 0 .. WRLEN/32-1
        WR[wd]_32i+31..32i ← WR[ws]_32i+31..32i - t
    endfor

SUBVI.D
    t ← 0^59 || u5_4..0
    for i in 0 .. WRLEN/64-1
        WR[wd]_64i+63..64i ← WR[ws]_64i+63..64i - t
    endfor
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception.




**Format:** `VSHF.df`

`VSHF.B wd,ws,wt` **MSA**

`VSHF.H wd,ws,wt` **MSA**

`VSHF.W wd,ws,wt` **MSA**

`VSHF.D wd,ws,wt` **MSA**

**Purpose:** Vector Data Preserving Shuffle

Vector elements selective copy based on the control vector preserving the input data vectors.

**Description:** `wd ← vector_shuffle(control(wd), ws, wt)`

The vector shuffle instructions selectively copy data elements from the concatenation of vectors *ws* and *wt* into vector *wd* based on the corresponding control element in *wd*.

The least significant 6 bits in *wd* control elements modulo the number of elements in the concatenated vectors *ws*, *wt* specify the index of the source element. If bit 6 or bit 7 is 1, there will be no copy, but rather the destination element is set to 0.

The operands and results are values in integer data format *df*.

**Restrictions:**

No data-dependent exceptions are possible.

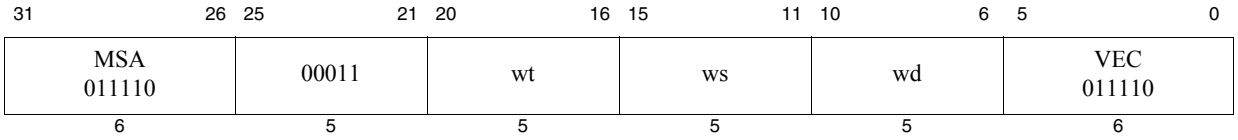
**Operation:**

```
VSHF.B
    v ← WR[ws] || WR[wt]
    for i in 0 .. WRLEN/8-1
        k ← WR[wd]8i+5..8i mod (WRLEN/4)
        if WR[wd]8i+7..8i+6 ≠ 0 then
            WR[wd]8i+7..8i ← 0
        else
            WR[wd]8i+7..8i ← v8k+7..8k
        endif
    endfor

VSHF.H
    v ← WR[ws] || WR[wt]
    for i in 0 .. WRLEN/16-1
        k ← WR[wd]16i+5..16i mod (WRLEN/8)
        if WR[wd]16i+7..16i+6 ≠0 then
            WR[wd]16i+15..16i ← 0
        else
            WR[wd]16i+15..16i ← v16k+15..16k
        endif
    endfor

VSHF.W
    v ← WR[ws] || WR[wt]
    for i in 0 .. WRLEN/32-1
        k ← WR[wd]32i+5..32i mod (WRLEN/16)
```

```
        if WR[wd]32i+7..32i+6 ≠ 0 then
            WR[wd]32i+31..32i ← 0
        else
            WR[wd]32i+31..32i ← v32k+31..32k
        endif
    endfor

VSHF.D
    v ← WR[ws] || WR[wt]
    for i in 0 .. WRLEN/64-1
        k ← WR[wd]64i+5..64i mod (WRLEN/32)
        if WR[wd]64i+7..64i+6 ≠ 0 then
            WR[wd]64i+63..64i ← 0
        else
            WR[wd]64i+63..64i ← v64k+63..64k
        endif
    endfor
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception.




**Format:** `XOR.V`
`XOR.V wd,ws,wt` **MSA**

**Purpose:** Vector Logical Exclusive Or

Vector by vector logical exclusive or.

**Description:** `wd ← ws XOR wt`

Each bit of vector *ws* is combined with the corresponding bit of vector *wt* in a bitwise logical XOR operation. The result is written to vector *wd*.

The operands and results are bit vector values.

**Restrictions:**

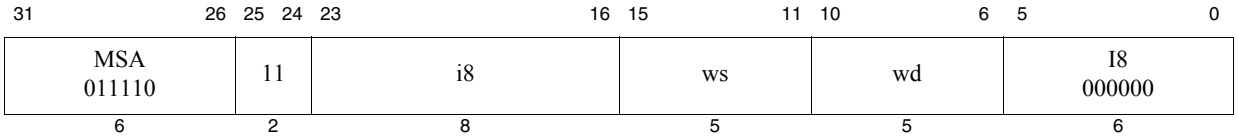
No data-dependent exceptions are possible.

**Operation:**

```
WR[wd] ← WR[ws] xor WR[wt]
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception.




**Format:** `XORI.B`
`XORI.B wd,ws,i8` **MSA**

**Purpose:** Immediate Logical Exclusive Or

Immediate by vector logical exclusive or.

**Description:** `wd[i] ← ws[i] XOR i8`

Each byte element of vector *ws* is combined with the 8-bit immediate *i8* in a bitwise logical XOR operation. The result is written to vector *wd*.

The operands and results are values in integer byte data format.

**Restrictions:**

No data-dependent exceptions are possible.

**Operation:**

```
for i in 0 .. WRLEN/8-1
  WR[wd]8i+7..8i ← WR[ws]8i+7..8i xor i87..0
endfor
```

**Exceptions:**

Reserved Instruction Exception, MSA Disabled Exception.

*Appendix A*

# Vector Registers Partitioning

MSA allows for multi-threaded implementations with fewer than 32 physical vector registers per hardware thread context. The thread contexts have access to as many vector registers as needed. When the hardware runs out of physical registers, the OS re-schedules the running threads or processes to accommodate for the pending requests.

The OS is responsible for saving and restoring the vector registers on software context switching. The actual mapping of the physical registers to the thread contexts is managed by the hardware itself and it is totally invisible to the software.

An overview of the this process is presented in the following sections. The hardware/software interface used for vector register allocation and software context switching relies on the MSA control registers and the MSA Access Disabled Exception, all described in Section 3.4 "MSA Control Registers" and Section 3.5 "Exceptions".

## A.1 Vector Registers Mapping

Let's assume an implementation with 4 hardware thread contexts $tc_0$, …, $tc_3$, and 64 physical vector registers $pv_0$, …, $pv_{63}$. Each hardware thread context has its own set of MSA control registers.

The hardware maintains a look-up table with the mapping of the 64 physical registers to any of the architecturally defined 32 vector registers W0, …, W31 usable from within the 4 hardware thread contexts. Hypothetically, the look-up table could be as shown in Table A.1.

**Table A.1 Physical-to-Thread Context Vector Register Mapping (Hardware Internal)**

| Physical Register | Hardware Thread Context | Architecture Register |
|---|---|---|
| $pv_0$ | $tc_3$ | W5 |
| $pv_1$ | $tc_3$ | W0 |
| $pv_2$ | *none* | *N/A* |
| $pv_4$ | $tc_0$ | W2 |
| … | … | … |
| $pv_{63}$ | *none* | *N/A* |

The OS grants a vector register to a hardware thread context by writing the register index to *MSAMap*. The successful mapping is confirmed in *MSAAccess*. For example, on writing 1 to *MSAMap*, the hardware finds a free physical

register, maps it to W1 for $tc_0$, and updates its internal look-up table (see Table A.2). Now that the context $tc_0$ already using W2 is being granted access to vector register is W1, the $tc_0$ *MSAAccess* control register changes from 0x00000004 (only $MSAAccess_{W2}$ bit set) to 0x00000006 (now $MSAAccess_{W2}$ and $MSAAccess_{W1}$ bits are set).

If the hardware runs out of physical vector registers to map, the *MSAAccess* does not change. To confirm the availability, the OS should read back and check *MSAAccess*.

**Table A.2 Updated Physical-to-Thread Context Vector Register Mapping (Hardware Internal)**

| Physical Register | Hardware Thread Context | Architecture Register |
|---|---|---|
| $pv_0$ | $tc_3$ | W5 |
| $pv_1$ | $tc_3$ | W0 |
| $pv_2$ | $tc_0$ | W1[1] |
| $pv_4$ | $tc_0$ | W2 |
| … | … | … |
| $pv_{63}$ | *none* | *N/A* |

1. Updated entry.

## A.2 Saving/Restoring Vector Registers on Context Switch

Using the above hardware implementation, i.e. 4 thread contexts $tc_0$, …, $tc_3$, and 64 physical vector registers $pv_0$, …, $pv_{63}$, the OS manages the context switching for a set of software threads, $s_0$, …, $s_{10}$, $s_{11}$, $s_{12}$, … Two look-up tables are used for this purpose: one with the status of the software context mapping and previously saved vector registers (Table A.3) and the second with the vector register usage for each software thread (Table A.4).

Table A.3 and Table A.4 show software thread $s_{10}$ on thread context $tc_0$ using vector register W2. The other running thread is $s_{11}$ on $tc_3$ using W0 and W5. The hardware view of this configuration has been presented above in Table A.1. In Table A.3, thread $s_{12}$ is waiting to be scheduled and has vector register W1 saved from a previous run.

**Table A.3 Context Mapping Table (OS Internal)**

| Software Thread | Hardware Thread Context | Status | Saved Registers (Hex Mask) | Saved Registers (Register List) |
|---|---|---|---|---|
| $s_{10}$ | $tc_0$ | running on | 0x00000000 | none |
| $s_{11}$ | $tc_3$ | running on | 0x00000000 | none |
| $s_{12}$ | *N/A* | waiting | 0x00000002 | W1 |

**Table A.4 Register Usage Table (OS Internal)**

| Software Thread | Hardware Thread Context | *MSAAccess* (Hex Mask) | *MSAAccess* (Register List) |
|---|---|---|---|
| $s_{10}$ | $tc_0$ | 0x00000004 | W2 |
| $s_{11}$ | $tc_3$ | 0x00000021 | W0, W5 |

Let's suppose there is context switch between $s_{10}$ and $s_{12}$ on $tc_0$. What the OS does is to start running $s_{12}$ on $tc_0$ without changing the current $tc_0$ *MSAAccess*, but setting in *MSASave* all the bits set in either *MSAAccess* or in the $s_{12}$ saved registers mask. Therefore *MSASave* has two bits set: $MSASave_{W2}$ and $MSASave_{W1}$, which allows for saving W2 register used by $s_{10}$ and restoring W1 register already saved for $s_{12}$ when this register is requested.

If the first MSA instruction $s_{12}/tc_0$ runs writes vector register W2 and reads vector register W1, the hardware sets $MSARequest_{W1}$, $MSARequest_{W2}$ and signals the MSA Access Disabled Exception. The exception is signaled because W2 needs to be saved, i.e. $MSASave_{W2}$ is set, and W1 is not available i.e. $MSAAccess_{W1}$ is clear. Then, the OS will take the following actions:

- Save W2 because $MSASave_{W2}$ is set. From the register usage Table A.4 it is known that $tc_0$/W2 belongs to $s_{10}$. Saving W2 requires a vector store followed by setting bit 2 in Saved Registers Mask of $s_{10}$, and clearing the $MSASave_{W2}$.
- Request a new physical vector register for W1 by writing 1 to *MSAMap*.
- Restore the previous W1 used by $s_{12}$ according to the Saved Registers Mask in Table A.3. Restoring W1 requires a vector load followed by clearing $MSASave_{W1}$. Because W1 has been written, the hardware will set $MSAModify_{W1}$.
- Clear $MSAModify_{W1}$ because the restored W1 is not changed with respect of the saved value. In this context, the $s_{12}$ Saved Registers Mask bit W1 is still relevant and should be preserved as set.

Table A.5 and Table A.6 show the software context mapping / saved registers and the vector register usage look-up tables after these updates.

**Table A.5 Updated Context Mapping Table (OS Internal)**

| Software Thread | Hardware Thread Context | Status | Saved Registers (Hex Mask) | Saved Registers (Register List) |
|---|---|---|---|---|
| $s_{10}$ | N/A | waiting | 0x00000004 | W2[1] |
| $s_{11}$ | $tc_3$ | running on | 0x00000000 | none |
| $s_{12}$ | $tc_0$ | running on | 0x00000002 | W1 |

1. Updated entry.

**Table A.6 Updated Register Usage Table (OS Internal)**

| Software Thread | Hardware Thread Context | *MSAAccess* (Hex Mask) | *MSAAccess* (Register List) |
|---|---|---|---|
| $s_{11}$ | $tc_3$ | 0x00000021 | W0, W5 |
| $s_{12}$ | $tc_0$ | 0x00000006 | W1, W2[1] |

1. Updated entry, $s_{10}$ changed to $s_{12}$.

## A.3 Re-allocating Physical Vector Registers

A physical register is mapped to a thread context/architecture register by writing the architecture register index to *MSAMap*. It is not relevant if the software knows what the particular mapping is — it can always access the same register from the same hardware thread context.

Physical vector registers re-allocation from one software thread to another on the same thread context (intra re-allocation) is done by setting the corresponding bits in the *MSASave* control register. If the new software thread starts with *MSASave* being identical to *MSAAccess*, it is guaranteed all vector registers used by the new software thread are properly saved/restored. An example of this procedure is presented above in Section A.2 "Saving/Restoring Vector Registers on Context Switch".

Inter-thread contexts physical vector registers re-allocation (between different hardware thread contexts), mandates the owner thread context to save all the registers intended for re-allocation and unmap them by writing the corresponding indexes to *MSAUnmap*. To exemplify, let's start from the configuration shown in Table A.5 / Table A.6 (OS view) and Table A.2 (hardware view). If the software decides to free up vector register W0 on $tc_3$ when re-scheduling $s_{11}$, then it saves W0, marks W0 as saved for $s_{11}$, and writes 0 to *MSAUnmap*. Then, the hardware will mark $pv_1$, i.e. the hypothetical mapping in Table A.2 used for W0/$tc_3$, as free. In a different thread context, let's say $tc_1$, the software could now map a new vector register, e.g. W9, and if the hardware decides $pv_1$ is the next free register, $pv_1$ will be used by $tc_1$ for W9.

## A.4 Heuristic for Vector Register Allocation

The performance of a multithreaded MSA implementation with less than 32 vector registers per thread context depends the actual register usage at run-time and the OS scheduling strategy.

In a typical application, one software thread might use lots of vector registers for longer time, while the other threads sporadically use very few. The OS could schedule the most demanding software thread on the same thread context, while time-sharing another context for the software threads with a lighter usage pattern.

*Appendix B*

# Revision History

| Revision | Date | Description |
|---|---|---|
| 1.00 | December 12, 2012 | • MIPS Architecture Release 5. |
| 1.01 | February 8, 2013 | • Signaling NaN definition, non-trapping exception pseudocode clarification.<br>• LDX/STX pseudocode typo fix.<br>• FLOG2 description clarification.<br>• Typo fix for 64-bit GPR-based instructions.<br>• Reserved df/n values for elements outside the 128-bit wide vector registers.<br>• Specified WRLEN constant to be 128.<br>• 3RF opcode table H/W vs. W/D typo fixed.<br>• Specified NaN propagation rule.<br>• FMADD/FMSUB signals Invalid for infinity * 0.<br>• CTCMSA/CFCMSA signal Coprocessor 0 Unusable exception for privileged MSA control registers<br>• MSA instruction can not be executed when FPU is usable and operates with floating-point registers in 32-bit mode.<br>• FTQ signals the Overflow exception for out of range numeric operands. |
| 1.02 | March 4, 2013 | • Reset state for MSAEn bit and MSA Access, Save, Modify and Request control registers is zero.<br>• Added new instructions: INSVE, FRCP, and FRSQRT instructions.<br>• Specified new flush to zero control bits.<br>• Clarified the effects of changing FR from 0 to 1 and from 1 to 0. |
| 1.03 | March 8, 2013 | • Specified the effect of FPR high read/write operations on the vector registers.<br>• Removed unused VECS5 instruction format. |
| 1.04 | May 31, 2013 | • Fixed NX mode description to specify that the output is always a signaling NaN value for any floating-point exception detected when NX is set.<br>• Clarified address calculation for load/store instructions with no alignment restrictions.<br>• Flush to zero is controlled with one bit (FS) for both subnormal input operands and tiny non-zero results.<br>• Clarified subnormal input operands flush to zero in compare instructions.<br>• FPR registers are **UNPREDICTABLE** after changing FR from 0 to 1 and from 1 to 0.<br>• Explicit MIPS Architecture Release 5 and FPU NAN2008/ABS2008 requirements.<br>• Renamed INSV to INSERT, SUBSS_U to SUBSUU_S, and SUBUS_S to SUBSUS_U.<br>• New instructions (FTRUNC_S, FTRUNC_U) for floating-point to integer truncation.<br>• New instructions for shift right with rounding (SRAR, SRARI. SRLR, SRLRI) and horizontal add/sub (HADD_S, HADD_U, HSUB_S, HSUB_U).<br>• Eliminated redundant floating point compare instructions FCGT, FSGT, FCGE, FSGE.<br>• New floating point compare instructions (FCAF, FSAF, FCUEQ, FSUEQ, FCULT, FSULT, FCULE, FSULE, FSUN, FCOR, FSOR, FCUNE, FSUNE).<br>• Opcode changes for FCNE, FSNE, MUL_Q, MULR_Q, MADD_Q, MADDR_Q, MSUB_Q, MSUBR_Q.<br>• Defined floating-point registers access in the context of vector registers partitioning.<br>• Load/store pseudocode update. |

| Revision | Date | Description |
|---|---|---|
| 1.05 | June 21, 2013 | • Template update to change MIPS logo and legal text.<br>• Flush to zero (FS) does not apply to 16-bit float data used by format conversion instructions FEXDO, FEXUPL, and FEXUPR and to non arithmetic instruction FCLASS.<br>• Load/store instructions are atomic at the element level and do not guarantee any ordering among elements.<br>• Defined reserved fields as R0: read as zero and must be written as zero.<br>• Clarified SLD/SLDI register layout and data format.<br>• FRCP and FRSQRT clarifications regarding Underflow, Overflow, and Inexact signaling. |
| 1.06 | August 6, 2013 | • Missing immediate instructions and FMSUB added to the Instruction Set Summary.<br>• Explicitly defined i8 immediates as 8-bit values where the sign is not relevant.<br>• Typos fixed for source and destination registers in VSHF.W and COPY_S/U pseudocode.<br>• COPY_S/U.D and INSERT.D are MIPS64 instructions. Updated ELM Instruction Format table accordingly.<br>• Added “ordered” text to the ordered floating-point compare instructions.<br>• Typo fixed in mulx_s/u pseudocode for bit selection.<br>• Changed MSA MIPS32 AFP document class to 2B.<br>• The default value for Underflow is the rounded result based on the rounding mode.<br>• Approximate reciprocal instructions FRCP and FRSQRT signal Inexact only for finite numerical operands. |
| 1.07 | October 2, 2013 | • Typo fixed in MSACSR Flags update pseudocode.<br>• Specified CTCMSA/CFCMSA reserved control registers behavior.<br>• Removed indexed load/store LDX/STX instructions.<br>• Introduced base architecture left-shift add LSA and DLSA instructions.<br>• LDI opcode changed.<br>• Load/store offsets are 10-bit values in data format units.<br>• Branch offsets are 16 bits.<br>• Added signaling to quiet NaN conversion rules.<br>• Corrections for fixed point multiply add/sub and signed-to-unsigned saturation pseudocode.<br>• Deleted the superfluous text for multiply add/sub NaN propagation as this case is no exception from the general left-to-right rule. |
| 1.09 | December 20, 2013 | • Fixed some typos in the instruction formats.<br>• Explicit referenced IEEE 2008 maxNum/maxNumMag and minNum/minNumMag in FMAX/FMAX_A and FMIN/FMIN_A.<br>• Typos fixed in FEXUPL description and FMAX_A pseudocode.<br>• FCLASS pseudocode typo fixed.<br>• FTQ signals both the Overflow and Inexact for values outside the range. |
| 1.10 | February 7, 2014 | • Expanded the text describing the NaN propagation rules.<br>• LD/ST descriptions show s10 offsets.<br>• Specified the flush-to-zero exception signaling for approximate reciprocal instructions.<br>• Reciprocal instructions FRCP and FRSQRT comply with the IEEE rules. |
| 1.11 | April 8, 2014 | • Higher vector register bits are **UNPREDICTABLE** after writing scalar floating-point values.<br>• Reserved MSA opcodes generate MSA Disabled exception.<br>• Specified that the assembler syntax for the LD/ST offset is in bytes.<br>• Neither the base address nor the calculated effective LD/ST address have any alignment restrictions. |
| 1.12 | February 3, 2016 | • COPY_U.D removed from MSA64.<br>• Replaced `u2` with `sa` in the LSA and DLSA descriptions.<br>• Load/store atomicity is guaranteed only if the address is element aligned.<br>• Fixed FFQL/FFQR scaling typo. |