MIPS64™ Architecture for Programmers
Volume IV-c: The MIPS-3D™
Application-Specific Extension to the MIPS64™
Architecture

Document Number: MD00099
Revision 1.11
March 12, 2001

MIPS Technologies, Inc.
1225 Charleston Road
Mountain View, CA 94043-1353
Table of Contents

Chapter 1 About This Book ......................................................................................................................................................... 1
  1.1 Typographical Conventions ................................................................................................................................................. 1
    1.1.1 Italic Text ........................................................................................................................................................................... 1
    1.1.2 Bold Text ............................................................................................................................................................................... 1
    1.1.3 Courier Text ......................................................................................................................................................................... 1
  1.2 UNPREDICTABLE and UNDEFINED ............................................................................................................................. 2
    1.2.1 UNPREDICTABLE ............................................................................................................................................................... 2
    1.2.2 UNDEFINED ......................................................................................................................................................................... 2
  1.3 Special Symbols in Pseudocode Notation ........................................................................................................................ 2
  1.4 For More Information .......................................................................................................................................................... 5

Chapter 2 Guide to the Instruction Set ..................................................................................................................................... 7
  2.1 Understanding the Instruction Fields ............................................................................................................................. 7
    2.1.1 Instruction Fields ............................................................................................................................................................... 8
    2.1.2 Instruction Descriptive Name and Mnemonic .................................................................................................................. 9
    2.1.3 Format Field ......................................................................................................................................................................... 9
    2.1.4 Purpose Field ....................................................................................................................................................................... 10
    2.1.5 Description Field ............................................................................................................................................................... 10
    2.1.6 Restrictions Field ............................................................................................................................................................... 10
    2.1.7 Operation Field ................................................................................................................................................................. 11
    2.1.8 Exceptions Field ............................................................................................................................................................... 11
    2.1.9 Programming Notes and Implementation Notes Fields ................................................................................................. 11
  2.2 Operation Section Notation and Functions .................................................................................................................. 12
    2.2.1 Instruction Execution Ordering ....................................................................................................................................... 12
    2.2.2 Pseudocode Functions ....................................................................................................................................................... 12
  2.3 Op and Function Subfield Notation ........................................................................................................................... 21
  2.4 FPU Instructions ............................................................................................................................................................... 21

Chapter 3 MIPS-3D™ Application-Specific Extension to the MIPS64™ Architecture ......................................................... 23
  3.1 Base Architecture Requirements ..................................................................................................................................... 23
  3.2 Software Detection of the ASE ........................................................................................................................................ 23
  3.3 MIPS-3D Overview ............................................................................................................................................................ 23
  3.4 Instruction Bit Encoding ....................................................................................................................................................... 24

Chapter 4 The MIPS-3D™ ASE Instruction Set .................................................................................................................. 27
  4.1 MIPS-3D Instruction Descriptions ............................................................................................................................. 27

Appendix A Revision History ................................................................................................................................................ 51
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2-1</td>
<td>Example of Instruction Description</td>
<td>8</td>
</tr>
<tr>
<td>2-2</td>
<td>Example of Instruction Fields</td>
<td>9</td>
</tr>
<tr>
<td>2-3</td>
<td>Example of Instruction Descriptive Name and Mnemonic</td>
<td>9</td>
</tr>
<tr>
<td>2-4</td>
<td>Example of Instruction Format</td>
<td>9</td>
</tr>
<tr>
<td>2-5</td>
<td>Example of Instruction Purpose</td>
<td>10</td>
</tr>
<tr>
<td>2-6</td>
<td>Example of Instruction Description</td>
<td>10</td>
</tr>
<tr>
<td>2-7</td>
<td>Example of Instruction Restrictions</td>
<td>11</td>
</tr>
<tr>
<td>2-8</td>
<td>Example of Instruction Operation</td>
<td>11</td>
</tr>
<tr>
<td>2-9</td>
<td>Example of Instruction Exception</td>
<td>11</td>
</tr>
<tr>
<td>2-10</td>
<td>Example of Instruction Programming Notes</td>
<td>12</td>
</tr>
<tr>
<td>2-11</td>
<td>COP_LW Pseudocode Function</td>
<td>13</td>
</tr>
<tr>
<td>2-12</td>
<td>COP_LD Pseudocode Function</td>
<td>13</td>
</tr>
<tr>
<td>2-13</td>
<td>COP_SW Pseudocode Function</td>
<td>13</td>
</tr>
<tr>
<td>2-14</td>
<td>COP_SD Pseudocode Function</td>
<td>14</td>
</tr>
<tr>
<td>2-15</td>
<td>AddressTranslation Pseudocode Function</td>
<td>14</td>
</tr>
<tr>
<td>2-16</td>
<td>LoadMemory Pseudocode Function</td>
<td>15</td>
</tr>
<tr>
<td>2-17</td>
<td>StoreMemory Pseudocode Function</td>
<td>15</td>
</tr>
<tr>
<td>2-18</td>
<td>Prefetch Pseudocode Function</td>
<td>16</td>
</tr>
<tr>
<td>2-19</td>
<td>ValueFPR Pseudocode Function</td>
<td>17</td>
</tr>
<tr>
<td>2-20</td>
<td>StoreFPR Pseudocode Function</td>
<td>18</td>
</tr>
<tr>
<td>2-21</td>
<td>SyncOperation Pseudocode Function</td>
<td>19</td>
</tr>
<tr>
<td>2-22</td>
<td>SignalException Pseudocode Function</td>
<td>19</td>
</tr>
<tr>
<td>2-23</td>
<td>NullifyCurrentInstruction Pseudocode Function</td>
<td>19</td>
</tr>
<tr>
<td>2-24</td>
<td>CoprocessorOperation Pseudocode Function</td>
<td>19</td>
</tr>
<tr>
<td>2-25</td>
<td>JumpDelaySlot Pseudocode Function</td>
<td>20</td>
</tr>
<tr>
<td>2-26</td>
<td>NotWordValue Pseudocode Function</td>
<td>20</td>
</tr>
<tr>
<td>2-27</td>
<td>FPConditionCode Pseudocode Function</td>
<td>20</td>
</tr>
<tr>
<td>2-28</td>
<td>SetFPConditionCode Pseudocode Function</td>
<td>21</td>
</tr>
</tbody>
</table>
List of Tables

Table 1-1: Symbols Used in Instruction Operation Statements ........................................................................................................ 3
Table 2-1: AccessLength Specifications for Loads/Stores .................................................................................................................... 16
Table 3-1: Instructions in the MIPS-3DTM ASE ........................................................................................................................................ 24
Table 3-2: Symbols Used in the Instruction Encoding Tables .................................................................................................................. 24
Table 3-3: MIPS-3D COP1 Encoding of rs Field ........................................................................................................................................... 25
Table 3-4: MIPS-3D COP1 Encoding of Function Field When rs=S ........................................................................................................ 25
Table 3-5: MIPS-3D COP1 Encoding of Function Field When rs=D ........................................................................................................ 25
Table 1: MIPS-3D COP1 Encoding of Function Field When rs=W or L .................................................................................................. 25
Table 3-6: MIPS-3D COP1 Encoding of Function Field When rs=PS .................................................................................................... 26
Chapter 1

About This Book

The MIPS64™ Architecture for Programmers Volume IV-c comes as a multi-volume set.

- Volume I describes conventions used throughout the document set, and provides an introduction to the MIPS64™ Architecture
- Volume II provides detailed descriptions of each instruction in the MIPS64™ instruction set
- Volume III describes the MIPS64™ Privileged Resource Architecture which defines and governs the behavior of the privileged resources included in a MIPS64™ processor implementation
- Volume IV-a describes the MIPS16™ Application-Specific Extension to the MIPS64™ Architecture
- Volume IV-b describes the MDMX™ Application-Specific Extension to the MIPS64™ Architecture
- Volume IV-c describes the MIPS-3D™ Application-Specific Extension to the MIPS64™ Architecture
- Volume IV-d describes the SmartMIPS™ Application-Specific Extension to the MIPS32™ Architecture and is not applicable to the MIPS64™ document set

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

1.3 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

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>←</td>
<td>Assignment</td>
</tr>
<tr>
<td>=, ≠</td>
<td>Tests for equality and inequality</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>x^y</td>
<td>A y-bit string formed by y copies of the single-bit value x</td>
</tr>
<tr>
<td>b#n</td>
<td>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 &quot;b#&quot; prefix is omitted, the default base is 10.</td>
</tr>
<tr>
<td>x_y..z</td>
<td>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.</td>
</tr>
<tr>
<td>+, −</td>
<td>2's complement or floating point arithmetic: addition, subtraction</td>
</tr>
<tr>
<td>∗, ×</td>
<td>2's complement or floating point multiplication (both used for either)</td>
</tr>
<tr>
<td>div</td>
<td>2's complement integer division</td>
</tr>
<tr>
<td>mod</td>
<td>2's complement modulo</td>
</tr>
<tr>
<td>/</td>
<td>Floating point division</td>
</tr>
<tr>
<td>&lt;</td>
<td>2's complement less-than comparison</td>
</tr>
<tr>
<td>&gt;</td>
<td>2's complement greater-than comparison</td>
</tr>
<tr>
<td>≤</td>
<td>2's complement less-than or equal comparison</td>
</tr>
<tr>
<td>≥</td>
<td>2's complement greater-than or equal comparison</td>
</tr>
<tr>
<td>nor</td>
<td>Bitwise logical NOR</td>
</tr>
<tr>
<td>xor</td>
<td>Bitwise logical XOR</td>
</tr>
<tr>
<td>and</td>
<td>Bitwise logical AND</td>
</tr>
<tr>
<td>or</td>
<td>Bitwise logical OR</td>
</tr>
<tr>
<td>GPRLEN</td>
<td>The length in bits (32 or 64) of the CPU general-purpose registers</td>
</tr>
<tr>
<td>GPR[x]</td>
<td>CPU general-purpose register x. The content of GPR[0] is always zero.</td>
</tr>
<tr>
<td>FPR[x]</td>
<td>Floating Point operand register x</td>
</tr>
<tr>
<td>FCC[CC]</td>
<td>Floating Point condition code CC. FCC[0] has the same value as COC[1].</td>
</tr>
<tr>
<td>FPR[x]</td>
<td>Floating Point (Coprocessor unit 1), general register x</td>
</tr>
<tr>
<td>CPR[z,x,s]</td>
<td>Coprocessor unit z, general register x, select s</td>
</tr>
<tr>
<td>CCR[z,x]</td>
<td>Coprocessor unit z, control register x</td>
</tr>
<tr>
<td>COC[z]</td>
<td>Coprocessor unit z condition signal</td>
</tr>
<tr>
<td>Xlat[x]</td>
<td>Translation of the MIPS16 GPR number x into the corresponding 32-bit GPR number</td>
</tr>
<tr>
<td>BigEndianMem</td>
<td>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.</td>
</tr>
</tbody>
</table>
Table 1-1 Symbols Used in Instruction Operation Statements

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>BigEndianCPU</td>
<td>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).</td>
</tr>
<tr>
<td>ReverseEndian</td>
<td>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 (SRRE and User mode).</td>
</tr>
<tr>
<td>LLbit</td>
<td>Bit of virtual state used to specify operation for instructions that provide atomic read-modify-write. LLbit is set when a linked load occurs; it is tested and cleared 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.</td>
</tr>
<tr>
<td>I+, I-n+; I-n</td>
<td>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. 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 labelled 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.</td>
</tr>
<tr>
<td>PC</td>
<td>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 MIPS16 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.</td>
</tr>
<tr>
<td>PABITS</td>
<td>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^{36}) bytes.</td>
</tr>
<tr>
<td>SEGBITS</td>
<td>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^{40}) bytes.</td>
</tr>
<tr>
<td>FP32RegistersMode</td>
<td>Indicates whether the FPU has 32-bit or 64-bit floating point registers (FPRs). In MIPS32, the FPU has 32 32-bit FPRs in which 64-bit data types are stored in even-odd pairs of FPRs. In MIPS64, the FPU has 32 64-bit FPRs in which 64-bit data types are stored in any FPR.</td>
</tr>
<tr>
<td></td>
<td>In MIPS32 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.</td>
</tr>
<tr>
<td>InstructionInBranchDelaySlot</td>
<td>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.</td>
</tr>
<tr>
<td>SignalException(exception, argument)</td>
<td>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.</td>
</tr>
</tbody>
</table>
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

Comments or questions on the MIPS64™ Architecture or this document should be directed to

Director of MIPS Architecture
MIPS Technologies, Inc.
1225 Charleston Road
Mountain View, CA 94043

or via E-mail to architecture@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:

- “Instruction Fields” on page 8
- “Instruction Descriptive Name and Mnemonic” on page 9
- “Format Field” on page 9
- “Purpose Field” on page 10
- “Description Field” on page 10
- “Restrictions Field” on page 10
- “Operation Field” on page 11
- “Exceptions Field” on page 11
- “Programming Notes and Implementation Notes Fields” on page 11
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](image)

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](image)

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.

![Format: ADD rd, rs, rt](image)

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. If the instruction was originally defined in the MIPS I through MIPS V levels of the architecture, that information is enclosed in parentheses.

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.

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

*Figure 2-5 Example of Instruction Purpose*

### 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.

**Description:** \(rd \leftarrow rs + rt\)

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 \(rd\)

*Figure 2-6 Example of Instruction Description*

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)
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.

Figure 2-7 Example of Instruction Restrictions

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.

Operation:

```pascal
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 temp 32 ≠ temp 31 then
    SignalException(IntegerOverflow)
else
    GPR[rd] ← sign_extend(temp 31..0)
endif
```

Figure 2-8 Example of Instruction Operation

See Section 2.2, "Operation Section Notation and Functions" on page 12 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.

Exceptions:
- Integer Overflow

Figure 2-9 Example of Instruction Exception

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

2.1.9 Programming Notes and Implementation Notes Fields
Chapter 2 Guide to the Instruction Set

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.

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

Figure 2-10 Example of Instruction Programming Notes

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 12
• “Pseudocode Functions” on page 12

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 12
• “Load Memory and Store Memory Functions” on page 14
• “Access Functions for Floating Point Registers” on page 16
• “Miscellaneous Functions” on page 18

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.
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

**Figure 2-11 COP_LW Pseudocode Function**

**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 \( rt \).

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

**Figure 2-12 COP_LD Pseudocode Function**

**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 \( rt \).

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

**Figure 2-13 COP_SW Pseudocode Function**

**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 \( rt \).
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

Figure 2-14 COP_SD Pseudocode Function

2.2.2.2 Load Memory and Store Memory 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 cache coherence algorithm, 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 cache coherence algorithm (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.

(pAddr, CCA) ← AddressTranslation (vAddr, IorD, LorS)
    /* pAddr: physical address */
    /* CCA: Cache Coherence Algorithm, 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

Figure 2-15 AddressTranslation Pseudocode Function

LoadMemory

The LoadMemory function loads a value from memory.
This action uses cache and main memory as specified in both the Cache Coherence Algorithm (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.

\[
\text{MemElem} \leftarrow \text{LoadMemory} (\text{CCA}, \text{AccessLength}, \text{pAddr}, \text{vAddr}, \text{IorD})
\]

/* 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: Cache Coherence Algorithm, the 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

Figure 2-16 LoadMemory Pseudocode Function

\textbf{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 Cache Coherence Algorithm (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.

\[
\text{StoreMemory} (\text{CCA}, \text{AccessLength}, \text{MemElem}, \text{pAddr}, \text{vAddr})
\]

/* CCA: Cache Coherence Algorithm, 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

Figure 2-17 StoreMemory Pseudocode Function

\textbf{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.

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

/* CCA: Cache Coherence Algorithm, 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

Figure 2-18 Prefetch Pseudocode Function

Table 2-1 lists the data access lengths and their labels for loads and stores.

### Table 2-1 AccessLength Specifications for Loads/Stores

<table>
<thead>
<tr>
<th>AccessLength Name</th>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>DOUBLEWORD</td>
<td>7</td>
<td>8 bytes (64 bits)</td>
</tr>
<tr>
<td>SEPTIBYTE</td>
<td>6</td>
<td>7 bytes (56 bits)</td>
</tr>
<tr>
<td>Sextibyte</td>
<td>5</td>
<td>6 bytes (48 bits)</td>
</tr>
<tr>
<td>Quintibyte</td>
<td>4</td>
<td>5 bytes (40 bits)</td>
</tr>
<tr>
<td>Word</td>
<td>3</td>
<td>4 bytes (32 bits)</td>
</tr>
<tr>
<td>Triplebyte</td>
<td>2</td>
<td>3 bytes (24 bits)</td>
</tr>
<tr>
<td>Halfword</td>
<td>1</td>
<td>2 bytes (16 bits)</td>
</tr>
<tr>
<td>Byte</td>
<td>0</td>
<td>1 byte (8 bits)</td>
</tr>
</tbody>
</table>

#### 2.2.2.3 Access Functions for Floating Point Registers

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.
value ← ValueFPR(fpr, fmt)

/* value: The formatted 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 ← UNPREDICTABLE32  ||  FPR[fpr]31..0
  D, UNINTERPRETED_DOUBLEWORD:
    if (FP32RegistersMode = 0)
      if (fpr0 0
        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)
      valueFPR ← UNPREDICTABLE
    else
      valueFPR ← FPR[fpr]
    endif
  DEFAULT:
    valueFPR ← UNPREDICTABLE
endcase
endfunction ValueFPR

Figure 2-19 ValueFPR Pseudocode Function

StoreFPR

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 (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 formatted 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 || value31..0
  D, UNINTERPRETED_DOUBLEWORD:
    if (FP32RegistersMode = 0)
      if (fpr = 0) then
        UNPREDICTABLE
      else
        FPR[fpr] ← UNPREDICTABLE || value31..0
        FPR[fpr+1] ← UNPREDICTABLE || value63..32
      endif
    else
      FPR[fpr] ← value
    endif
  L, PS, OB, QH:
    if (FP32RegistersMode = 0) then
      UNPREDICTABLE
    else
      FPR[fpr] ← value
    endif
endcase
endfunction StoreFPR

Figure 2-20 StoreFPR Pseudocode Function

2.2.2.4 Miscellaneous Functions

This section lists miscellaneous functions not covered in previous sections.

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 style occur in the same order for all processors.
SyncOperation(stype)

/* stype: Type of load/store ordering to perform. */
/* Perform implementation-dependent operation to complete the */
/* required synchronization operation */
endfunction SyncOperation

**Figure 2-21 SyncOperation Pseudocode Function**

*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.

SignalException(Exception, argument)

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

**Figure 2-22 SignalException Pseudocode Function**

*NullifyCurrentInstruction*

The *NullifyCurrentInstruction* function nullifies the current instruction.

The instruction is aborted. For branch-likely instructions, nullification kills the instruction in the delay slot during its execution.

NullifyCurrentInstruction()
endfunction NullifyCurrentInstruction

**Figure 2-23 NullifyCurrentInstruction Pseudocode Function**

*CoprocessorOperation*

The *CoprocessorOperation* function performs the specified Coprocessor operation.

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

**Figure 2-24 CoprocessorOperation Pseudocode Function**
JumpDelaySlot

The JumpDelaySlot function is used in the pseudocode for the four PC-relative instructions. 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.

\[
\text{JumpDelaySlot}(v\text{Addr})
\]

\[
/* vAddr: Virtual address */
\]

endfunction JumpDelaySlot

**Figure 2-25 JumpDelaySlot Pseudocode Function**

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.

\[
\text{result} \leftarrow \text{NotWordValue}(\text{value})
\]

\[
/* \text{result}: True if the value is not a correct sign-extended word value; */
/* False otherwise */
/* \text{value}: A 64-bit register value to be checked */
\]

\[
\text{NotWordValue} \leftarrow \text{value}_{63..32} \neq (\text{value}_{31})^{32}
\]

endfunction NotWordValue

**Figure 2-26 NotWordValue Pseudocode Function**

FPConditionCode

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

\[
\text{tf} \leftarrow \text{FPConditionCode}(cc)
\]

\[
/* \text{tf}: The value of the specified condition code */
/* \text{cc}: The Condition code number in the range 0..7 */
\]

\[
\text{if } cc = 0 \text{ then}
\]
\[
\text{FPConditionCode} \leftarrow \text{FCSR}_{23}
\]
\[
\text{else}
\]
\[
\text{FPConditionCode} \leftarrow \text{FCSR}_{24+cc}
\]
\[
\text{endif}
\]

endfunction FPConditionCode

**Figure 2-27 FPConditionCode Pseudocode Function**

SetFPConditionCode

The SetFPConditionCode function writes a new value to a specific floating point condition code.
SetFPConditionCode(cc)
    if cc = 0 then
        FCSR ← FCSR_{31..24} || tf || FCSR_{22..0}
    else
        FCSR ← FCSR_{31..25+cc} || tf || FCSR_{23+cc..0}
    endif
endfunction SetFPConditionCode

Figure 2-28 SetFPConditionCode Pseudocode Function

2.3 Op and Function Subfield Notation

In some instructions, the instruction subfields \textit{op} and \textit{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, \textit{op}=$\text{COP1}$ and \textit{function}=$\text{ADD}$. 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 \textit{fs}, \textit{ft}, immediate, and so on) are shown in lowercase. The instruction name (such as \textit{ADD}, \textit{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, \textit{rs}=$\text{base}$ 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 MIPS16 instructions.

See Section 2.3 , "Op and Function Subfield Notation" on page 21 for a description of the \textit{op} and \textit{function} subfields.
This chapter describes the purpose and key features of the MIPS-3D™ Application-Specific Extension (ASE) to the MIPS64™ Architecture.

3.1 Base Architecture Requirements

The MIPS-3D ASE requires the following base architecture support:

- **The MIPS64 Architecture**: The MIPS-3D ASE is not compatible with a MIPS32™ implementation.
- **The MIPS64 floating point option with all data types implemented**: The MIPS-3D ASE requires a floating point implementation that includes the single (S), double (D), word (W), long (L), and paired single (PS) datatypes.

3.2 Software Detection of the ASE

Software may determine if the MIPS-3D ASE is implemented by checking the state of the FP bit in the Config1 CP0 register to determine if floating is implemented. If this bit is set, software should then enable access to Coprocessor 1 by setting the CU1 bit in the Status register and checking the state of the 3D bit in the FIR CP1 control register.

3.3 MIPS-3D Overview

The MIPS-3D ASE comprises thirteen instructions added to the floating-point instruction set. These instructions are designed to improve the performance of graphics geometry code (triangle transform and lighting code) executed on the MIPS processor. Table 3-1 lists these thirteen instructions by function. Chapter 4, “The MIPS-3D™ ASE Instruction Set,” on page 27, describes these instructions in greater detail.

The table and instruction descriptions use the following notations for data formats:

- S for single data format (32 bits)
- D for double data format (64 bits)
- PS for paired-single data format (two singles in a 64-bit register)
- PL for paired-lower, the single value in bits 0-31 of the paired-single value in the 64-bit register
- PU for paired-upper, the single value in bits 32-63 of the paired-single value in the 64-bit register
- PW for paired-word data format (two words in a 64-bit register)
Chapter 3 MIPS-3D™ Application-Specific Extension to the MIPS64™ Architecture

3.4 Instruction Bit Encoding

Table 3-3 through Table 3-6 describe the encoding used for the MIPS-3D ASE. Table 3-2 describes the meaning of the symbols used in the tables. These tables only list the instruction encodings for the MIPS-3D instructions. See Volume I of this multi-volume set for a full encoding of all instructions.

Table 3-1 Instructions in the MIPS-3D™ ASE

<table>
<thead>
<tr>
<th>Type</th>
<th>Mnemonic</th>
<th>Valid Formats</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetic</td>
<td>ADDR</td>
<td>PS</td>
<td>Floating point reduction add</td>
</tr>
<tr>
<td></td>
<td>MULR</td>
<td>PS</td>
<td>Floating point reduction multiply</td>
</tr>
<tr>
<td></td>
<td>RECIP1</td>
<td>S, D, PS</td>
<td>Reciprocal first step with a reduced precision result</td>
</tr>
<tr>
<td></td>
<td>RECIP2</td>
<td>S, D, PS</td>
<td>Reciprocal second step (enroute to the full precision result)</td>
</tr>
<tr>
<td></td>
<td>RSQRT1</td>
<td>S, D, PS</td>
<td>Reciprocal square-root with a reduced precision result</td>
</tr>
<tr>
<td></td>
<td>RSQRT2</td>
<td>S, D, PS</td>
<td>Reciprocal square-root second step (enroute to the full precision result)</td>
</tr>
<tr>
<td>Format conversions</td>
<td>CVT.PS.PW</td>
<td>PW</td>
<td>Converts a pair of 32-bit fixed point integers to paired-single FP format</td>
</tr>
<tr>
<td></td>
<td>CVT.PW.PS</td>
<td>PS</td>
<td>Converts a paired-single FP format to a pair of 32-bit fixed point integers</td>
</tr>
<tr>
<td>Compare</td>
<td>CABS</td>
<td>S, D, PS</td>
<td>Magnitude compare of floating point numbers</td>
</tr>
<tr>
<td>Branch</td>
<td>BC1ANY2F</td>
<td></td>
<td>Branch if either one of the two specified (consecutive) condition codes is False</td>
</tr>
<tr>
<td></td>
<td>BC1ANY2T</td>
<td></td>
<td>Branch if either one of the two specified (consecutive) condition codes is True</td>
</tr>
<tr>
<td></td>
<td>BC1ANY4F</td>
<td></td>
<td>Branch if any one of the four specified (consecutive) condition codes is False</td>
</tr>
<tr>
<td></td>
<td>BC1ANY4T</td>
<td></td>
<td>Branch if any one of the four specified (consecutive) condition codes is True</td>
</tr>
</tbody>
</table>

Table 3-2 Symbols Used in the Instruction Encoding Tables

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>δ</td>
<td>(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.</td>
</tr>
<tr>
<td>ε</td>
<td>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.</td>
</tr>
</tbody>
</table>
### Table 3-3 MIPS-3D COP1 Encoding of rs Field

<table>
<thead>
<tr>
<th>rs</th>
<th>bits 23..21</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
</tr>
<tr>
<td>bits 25..24</td>
<td>000</td>
</tr>
<tr>
<td>0</td>
<td>00</td>
</tr>
<tr>
<td>1</td>
<td>01</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
</tr>
<tr>
<td>3</td>
<td>11</td>
</tr>
</tbody>
</table>

### Table 3-4 MIPS-3D COP1 Encoding of Function Field When rs=S

<table>
<thead>
<tr>
<th>function</th>
<th>bits 2..0</th>
</tr>
</thead>
<tbody>
<tr>
<td>bits 5..3</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>000</td>
</tr>
<tr>
<td>0</td>
<td>000</td>
</tr>
<tr>
<td>1</td>
<td>001</td>
</tr>
<tr>
<td>2</td>
<td>010</td>
</tr>
<tr>
<td>3</td>
<td>011</td>
</tr>
<tr>
<td>4</td>
<td>100</td>
</tr>
<tr>
<td>5</td>
<td>101</td>
</tr>
<tr>
<td>6</td>
<td>110</td>
</tr>
<tr>
<td>7</td>
<td>111</td>
</tr>
</tbody>
</table>

### Table 3-5 MIPS-3D COP1 Encoding of Function Field When rs=D

<table>
<thead>
<tr>
<th>function</th>
<th>bits 2..0</th>
</tr>
</thead>
<tbody>
<tr>
<td>bits 5..3</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>000</td>
</tr>
<tr>
<td>0</td>
<td>000</td>
</tr>
<tr>
<td>1</td>
<td>001</td>
</tr>
<tr>
<td>2</td>
<td>010</td>
</tr>
<tr>
<td>3</td>
<td>011</td>
</tr>
<tr>
<td>4</td>
<td>100</td>
</tr>
<tr>
<td>5</td>
<td>101</td>
</tr>
<tr>
<td>6</td>
<td>110</td>
</tr>
<tr>
<td>7</td>
<td>111</td>
</tr>
</tbody>
</table>

### Table 1 MIPS-3D COP1 Encoding of Function Field When rs=W or L

<table>
<thead>
<tr>
<th>function</th>
<th>bits 2..0</th>
</tr>
</thead>
<tbody>
<tr>
<td>bits 5..3</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>000</td>
</tr>
<tr>
<td>0</td>
<td>000</td>
</tr>
<tr>
<td>1</td>
<td>001</td>
</tr>
<tr>
<td>2</td>
<td>010</td>
</tr>
<tr>
<td>3</td>
<td>011</td>
</tr>
<tr>
<td>4</td>
<td>100</td>
</tr>
<tr>
<td>5</td>
<td>101</td>
</tr>
<tr>
<td>6</td>
<td>110</td>
</tr>
<tr>
<td>7</td>
<td>111</td>
</tr>
</tbody>
</table>

CVT.PS.PW &e
### Table 3-6 MIPS-3D COP1 Encoding of Function Field When \( rs = PS \)

<table>
<thead>
<tr>
<th>Function</th>
<th>Bits 5..3</th>
<th>Bits 2..0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 000</td>
<td>000</td>
<td>0</td>
</tr>
<tr>
<td>0 001</td>
<td>001</td>
<td>1</td>
</tr>
<tr>
<td>2 010</td>
<td>010</td>
<td>2</td>
</tr>
<tr>
<td>3 011</td>
<td>011</td>
<td>ADDR ( \varepsilon ) MULR ( \varepsilon ) RECIP2 ( \varepsilon ) RECIP1 ( \varepsilon ) RSQRT1 ( \varepsilon ) RSQRT2 ( \varepsilon )</td>
</tr>
<tr>
<td>4 100</td>
<td>100</td>
<td>CVT.PW.PS ( \varepsilon )</td>
</tr>
<tr>
<td>5 101</td>
<td>101</td>
<td>PLL.PS ( \varepsilon ) PLU.PS ( \varepsilon ) PUL.PS ( \varepsilon ) PUU.PS ( \varepsilon )</td>
</tr>
<tr>
<td>6 110</td>
<td>110</td>
<td>CABS.F ( \varepsilon ) CABS.UN ( \varepsilon ) CABS.EQ ( \varepsilon ) CABS.UEQ ( \varepsilon ) CABS.OLT ( \varepsilon ) CABS.ULT ( \varepsilon ) CABS.OLE ( \varepsilon ) CABS.ULE ( \varepsilon )</td>
</tr>
<tr>
<td>7 111</td>
<td>111</td>
<td>CABS.SF ( \varepsilon ) CABS.NGLE ( \varepsilon ) CABSSEQ ( \varepsilon ) CABS.NGL ( \varepsilon ) CABS.LT ( \varepsilon ) CABS.NGE ( \varepsilon ) CABS.LE ( \varepsilon ) CABS.NGT ( \varepsilon )</td>
</tr>
</tbody>
</table>
Chapter 4

The MIPS-3D™ ASE Instruction Set

4.1 MIPS-3D Instruction Descriptions

This chapter provides an alphabetic listing of the instructions listed in Table 3-1.
### Floating Point Reduction Add

<table>
<thead>
<tr>
<th>COP1</th>
<th>fmt</th>
<th>ft</th>
<th>fs</th>
<th>fd</th>
<th>ADDR.PS</th>
</tr>
</thead>
<tbody>
<tr>
<td>010001</td>
<td>10110</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>

**Format:** ADDR.PS fd, fs, ft

**Purpose:**
To perform a reduction add on two paired-single floating point values

**Description:**
$$fd.PL \leftarrow ft.PU + ft.PL; \quad fd.PU \leftarrow fs.PU + fs.PL$$

The paired-single values in FPR $ft$ are added together and the result put in the lower paired-single position of FPR $fd$. Similarly, the paired-single values in FPR $fs$ are added together and the result put in the upper paired-single position of FPR $fd$. The two results are calculated to infinite precision and rounded by using the current rounding mode in $FCSR$. The operands and result are values in format PS.

Any generated exceptions in the two independent adds are OR’ed together. *Cause* bits are ORed into the *Flag* bits if no exception is taken.

**Restrictions:**
The fields $fs$, $ft$, and $fd$ must specify FPRs valid for operands of type PS. If they are not valid, the result is UNPREDICTABLE.

The operands must be values in format PS. If they are not, the result is UNPREDICTABLE and the values in the operand FPRs become UNPREDICTABLE.

The result of ADDR.PS is UNPREDICTABLE if the processor is executing in 16 FP registers mode.

**Operation:**

- lower $\leftarrow$ ValueFPR($ft$, PS)$_{31..0}$ + ValueFPR($ft$, PS)$_{63..32}$
- upper $\leftarrow$ ValueFPR($fs$, PS)$_{31..0}$ + ValueFPR($fs$, PS)$_{63..32}$
- StoreFPR (fd, PS, upper $||$ lower)

**Exceptions:**
Coprocessor Unusable, Reserved Instruction

**Floating Point Exceptions:**
Unimplemented Operation, Invalid Operation, Overflow, Inexact, Underflow
Branch on Any of Two Floating Point Condition Codes False

**Format:** BC1ANY2F cc,offset

**Purpose:**
To test two consecutive floating point condition codes and do a PC-relative conditional branch.

**Description:** If CCn+1 = 0 or CCn = 0, then branch

An 18-bit signed offset (the 16-bit offset field shifted left 2 bits) is added to the address of the instruction following the branch (not the branch itself) in the branch delay slot to form a PC-relative effective target address. If either one of the two FP condition code bits CC is false (0), the program branches to the effective target address after the instruction in the delay slot is executed.

The CC specified must align to 2, so bit 18 must always be zero. For example, specifying a value of 4 will check if either one of CC5 or CC4 is 0 and branch accordingly. Specifying an illegally aligned CC will result in UNPREDICTABLE behavior.

An FP condition code is set by an FP compare instruction, C.cond.fmt and the MIPS-3D compare absolute instruction CABS.cond.fmt.

**Restrictions:**
Processor operation is UNPREDICTABLE if a branch, jump, ERET, DERET, or WAIT instruction is placed in the delay slot of a branch or jump.

**Operation:**
This operation specification is for the general Branch On Any Two Condition operation with the tf (true/false) as a variables. The individual instructions BC1ANY2F and BC1ANY2T have a specific values for tf.

I: condition ← (FPConditionCode(cc) = 0) or (FPConditionCode(cc+1) = 0)
   target_offset ← (offset15)GPRLEN-(16+2) || offset || 0

I+1: if condition then
     PC ← PC + target_offset
   endif
Branch on Any of Two Floating Point Condition Codes False, cont.  BC1ANY2F

Exceptions:
Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:
Unimplemented Operation

Programming Notes:
With the 18-bit signed instruction offset, the conditional branch range is ± 128 KBytes. Use jump (J) or jump register (JR) instructions to branch to addresses outside this range.
Branch on Any of Two Floating Point Condition Codes True

**Format:** BC1ANY2T cc, offset

**Purpose:**
To test two consecutive FP condition codes and do a PC-relative conditional branch

**Description:** If CC_{n+1} = 1 or CC_{n} = 1, then branch

An 18-bit signed offset (the 16-bit offset field shifted left 2 bits) is added to the address of the instruction following the branch (not the branch itself) in the branch delay slot to form a PC-relative effective target address. If either one of the two FP condition code bits CC is true (1), the program branches to the effective target address after the instruction in the delay slot is executed.

The CC specified must align to 2, so bit 18 must always be zero. For example, specifying a value of 2 will check if either one of CC_3 or CC_2 is 1 and branch accordingly. Specifying an illegally aligned CC will result in UNPREDICTABLE behavior.

An FP condition code is set by an FP compare instruction, C.cond.fmt and the MIPS-3D compare absolute instruction CABS.cond.fmt.

**Restrictions:**
Processor operation is UNPREDICTABLE if a branch, jump, ERET, DERET, or WAIT instruction is placed in the delay slot of a branch or jump.

**Operation:**
This operation specification is for the general Branch On Any Two Condition operation with the tf (true/false) as a variables. The individual instructions BC1ANY2F and BC1ANY2T have a specific values for tf.

\[
\begin{align*}
\text{I:} & \quad \text{condition} \leftarrow (\text{FPConditionCode}(cc) = 1) \text{ or } (\text{FPConditionCode}(cc+1) = 1) \\
& \quad \text{target_offset} \leftarrow (\text{offset}_{15})^{\text{GPRLEN}-16+2} || \text{offset} || 0^2 \\
\text{I+1:} & \quad \text{if condition then} \\
& \quad \text{PC} \leftarrow \text{PC} + \text{target_offset} \\
& \quad \text{endif}
\end{align*}
\]
### Branch on Any of Two Floating Point Condition Codes True, cont.

<table>
<thead>
<tr>
<th>Exception Type</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Exceptions:</strong></td>
</tr>
<tr>
<td>Coprocessor Unusable, Reserved Instruction</td>
</tr>
</tbody>
</table>

**Floating Point Exceptions:**
Unimplemented Operation

**Programming Notes:**
With the 18-bit signed instruction offset, the conditional branch range is ± 128 KBytes. Use jump (J) or jump register (JR) instructions to branch to addresses outside this range.
4.1 MIPS-3D Instruction Descriptions

Branch on Any of Four Floating Point Condition Codes False  

<table>
<thead>
<tr>
<th>Format:</th>
<th>BC1ANY4F cc,offset</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Purpose:</strong></td>
<td>To test four consecutive FP condition codes and do a PC-relative conditional branch</td>
</tr>
</tbody>
</table>

**Description:** If CCn+3 = 0 or CCn+2 = 0 or CCn+1 = 0 or CCn = 0, then branch

An 18-bit signed offset (the 16-bit offset field shifted left 2 bits) is added to the address of the instruction following the branch (not the branch itself) in the branch delay slot to form a PC-relative effective target address. If any of the four FP condition code bits CC is false (0), the program branches to the effective target address after the instruction in the delay slot is executed.

The CC specified must align to 4, so bits 18 and 19 must always be zero. For example, specifying a value of 0 will check if any one of CC3..0 is 0 and branch accordingly. Specifying an illegally aligned CC will result in UNPREDICTABLE behavior.

An FP condition code is set by an FP compare instruction, C.cond.fmt and the MIPS-3D compare absolute instruction CABS.cond.fmt.

**Restrictions:**
Processor operation is UNPREDICTABLE if a branch, jump, ERET, DERET, or WAIT instruction is placed in the delay slot of a branch or jump.

**Operation:**
This operation specification is for the general Branch On Any Four Condition operation with the tf (true/false) as a variables. The individual instructions BC1ANY4F and BC1ANY4T have a specific values for tf.

\[
I: \text{condition} \leftarrow \begin{cases} 
(FPConditionCode(cc) = 0) \lor \\
(FPConditionCode(cc+1) = 0) \lor \\
(FPConditionCode(cc+2) = 0) \lor \\
(FPConditionCode(cc+3) = 0)
\end{cases} \\
\text{target_offset} \leftarrow (offset_{15})^{GPRLEN-(16+2)} || offset || 0^2
\]

\[
I+1: \text{if condition then} \\
\text{PC} \leftarrow \text{PC} + \text{target_offset} \\
\text{endif}
\]
Branch on Any of Four Floating Point Condition Codes False, cont.  BC1ANY4F

**Exceptions:**
Coprocessor Unusable, Reserved Instruction

**Floating Point Exceptions:**
Unimplemented Operation

**Programming Notes:**
With the 18-bit signed instruction offset, the conditional branch range is ± 128 KBytes. Use jump (J) or jump register (JR) instructions to branch to addresses outside this range.
**Branch on Any of Four Floating Point Condition Codes True**

**Format:** \texttt{BC1ANY4T cc,offset} \quad \texttt{MIPS-3D}

**Purpose:**
To test four consecutive FP condition codes and do a PC-relative conditional branch

**Description:** If \( CC_{n+3} = 1 \) or \( CC_{n+2} = 1 \) or \( CC_{n+1} = 1 \) or \( CC_{n} = 1 \), then branch.

An 18-bit signed offset (the 16-bit \texttt{offset} field shifted left 2 bits) is added to the address of the instruction following the branch (not the branch itself) in the branch delay slot to form a PC-relative effective target address. If any of four FP condition code bits \( CC \) is true (1), the program branches to the effective target address after the instruction in the delay slot is executed.

The \( CC \) specified must align to 4, so bits 18 and 19 must always be zero. For example, specifying a value of 4 will check if any of the bits \( CC_{7..4} \) is 1 and branch accordingly. Specifying an illegally aligned \( CC \) will result in \texttt{UNPREDICTABLE} behavior.

An FP condition code is set by an FP compare instruction, \texttt{C.cond.fmt} and the MIPS-3D compare absolute instruction \texttt{CABS.cond.fmt}.

**Restrictions:**
Processor operation is \texttt{UNPREDICTABLE} if a branch, jump, ERET, DERET, or WAIT instruction is placed in the delay slot of a branch or jump.

**Operation:**
This operation specification is for the general Branch On Any Four Condition operation with the \( tf \) (true/false) as a variables. The individual instructions \texttt{BC1ANY4F} and \texttt{BC1ANY4T} have a specific values for \( tf \).

\begin{align*}
\text{I:} & \quad \text{condition} \leftarrow (\text{FPConditionCode}(cc) = 1) \text{ or } (\text{FPConditionCode}(cc+1) = 1) \text{ or } (\text{FPConditionCode}(cc+2) = 1) \text{ or } (\text{FPConditionCode}(cc+3) = 1) \\
& \quad \text{target\_offset} \leftarrow (\text{offset}_{15})^{\text{GPRLLEN-(16+2)}} \text{ || offset || } 0^2 \\
\text{I+1:} & \quad \text{if condition then} \\
& \quad \text{PC} \leftarrow \text{PC} + \text{target\_offset} \\
& \quad \text{endif}
\end{align*}
Branch on Any of Four Floating Point Condition Codes True, cont.

**Exceptions:**
Coprocessor Unusable, Reserved Instruction

**Floating Point Exceptions:**
Unimplemented Operation

**Programming Notes:**
With the 18-bit signed instruction offset, the conditional branch range is ± 128 KBytes. Use jump (J) or jump register (JR) instructions to branch to addresses outside this range.
### Floating Point Absolute Compare

<table>
<thead>
<tr>
<th>COP1</th>
<th>fmt</th>
<th>ft</th>
<th>fs</th>
<th>cc</th>
<th>A</th>
<th>FC</th>
<th>cond</th>
</tr>
</thead>
<tbody>
<tr>
<td>010001</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

**Format:**
- CABS.cond.S cc,fs,ft
- CABS.cond.D cc,fs,ft
- CABS.cond.PS cc,fs,ft

**Purpose:**
To compare FP values and record the boolean result in one or more condition codes

**Description:**
\[ cc \leftarrow fs \text{ compare} \_\text{absolute} \_\text{cond} \ ft \]

The absolute value in FPR \( fs \) is compared to the absolute value in FPR \( ft \); the values are in format \( fmt \). The comparison is exact and neither overflows nor underflows.

If the comparison specified by \( \text{cond} \) is true for the operand values, the result is true; otherwise, the result is false. If no exception is taken, the result is written into condition code \( CC \); true is 1 and false is 0.

CABS.cond.PS compares the upper and lower halves of FPR \( fs \) and FPR \( ft \) independently and writes the results into condition codes \( CC +1 \) and \( CC \) respectively. The CC number must be even. If the number is not even the operation of the instruction is **UNPREDICTABLE**.

See the description of the C.cond.fmt instruction in Volume II of this multi-volume set for a complete description of the cond value and the behavior of the compare instruction.

**Restrictions:**
The fields \( fs \) and \( ft \) must specify FPRs valid for operands of type \( fmt \); if they are not valid, the result is **UNPREDICTABLE**.

The operands must be values in format \( fmt \); if they are not, the result is **UNPREDICTABLE** and the value of the operand FPRs becomes **UNPREDICTABLE**.

The result of CABS.cond.PS is **UNPREDICTABLE** if the processor is executing in 16 FP registers mode, or if the condition code number is odd.
Operation:

\[
\text{if } \text{SNaN(ValueFPR(fs, fmt)) or SNaN(ValueFPR(ft, fmt)) or}
\text{QNaN(ValueFPR(fs, fmt)) or QNaN(ValueFPR(ft, fmt)) then}
\]
less ← false
equal ← false
unordered ← true

\[
\text{if (SNaN(ValueFPR(fs,fmt)) or SNaN(ValueFPR(ft,fmt))) or}
\text{(cond\_3 and (QNaN(ValueFPR(fs,fmt)) or QNaN(ValueFPR(ft,fmt)))) then}
\]
\[
\text{SignalException(InvalidOperation)}
\]
endif
else

\[
\text{less ← AbsoluteValue(ValueFPR(fs, fmt)) <fmt}
\]

\[
\text{AbsoluteValue(ValueFPR(ft, fmt))}
\]

\[
\text{equal ← AbsoluteValue(ValueFPR(fs, fmt)) =fmt}
\]

\[
\text{AbsoluteValue(ValueFPR(ft, fmt))}
\]

\[
\text{unordered ← false}
\]
endif

\[
\text{condition ← (cond\_2 and less) or (cond\_1 and equal)}
\]
\[
\text{or (cond\_0 and unordered)}
\]

SetFPConditionCode(cc, condition)

For CABS.cond.PS, the pseudo code above is repeated for both halves of the operand registers, treating each half as an independent single-precision values. Exceptions on the two halves are logically ORed and reported together. The results of the lower half comparison are written to condition code CC; the results of the upper half comparison are written to condition code CC+1.

Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

Unimplemented Operation, Invalid Operation
Floating Point Convert Paired Single to Paired Word

**Format:** \texttt{CVT.PW.PS fd,fs} 

**Purpose:**
To convert a FP paired-single value to a pair of 32-bit fixed point words

**Description:**
\( fd.PU \leftarrow \text{convert\_and\_round}(fs.PU); \) \( fd.PL \leftarrow \text{convert\_and\_round}(fs.PL) \)

The values in FPR \( fs \), in format \( PS \), are converted to a pair of values in 32-bit word fixed point format and rounded according to the current rounding mode in \( FCSR \). The result is placed in FPR \( fd \). The conversions of the two halves are done independently.

When either source value is Infinity, NaN, or rounds to an integer outside the range \(-2^{31} \) to \( 2^{31} - 1 \), the result cannot be represented correctly, an IEEE Invalid Operation condition exists, and the Invalid Operation flag is set in the \( FCSR \). If the Invalid Operation Enable bit is set in the \( FCSR \), no result is written to \( fd \) and an Invalid Operation exception is taken immediately. Otherwise, the default result, \( 2^{31} - 1 \), is written to the correspond half of FPR \( fd \) which caused the exception.

**Restrictions:**
The fields \( fs \) and \( fd \) must specify valid FPRs---\( fs \) for type PS and \( fd \) for type PW. If they are not valid, the result is UNPREDICTABLE. The format of the data in the specified operand register \( fs \) must be a value in format PS; if it is not, the result is UNPREDICTABLE and the value in the operand FPR becomes UNPREDICTABLE.

The result of this instruction is UNPREDICTABLE if the processor is executing in 16 FP registers mode.

**Operation:**
\[
\begin{array}{l}
\text{StoreFPR}(fd, \text{PW},) \\
\quad \text{ConvertFmt(ValueFPR}(fs, \text{PS})_{63..32}, S, W) || \\
\quad \text{ConvertFmt(ValueFPR}(fs, \text{PS})_{31..0}, S, W)
\end{array}
\]
Floating Point Convert Paired Single to Paired Word (cont.)

**Exceptions:**
Coprocessor Unusable, Reserved Instruction

**Floating Point Exceptions**
Unimplemented Operation, Invalid Operation, Overflow, Inexact
Floating Point Convert Paired Word to Paired Single

**CVT.PS.PW**

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>COP1</td>
<td>fmt</td>
<td>0</td>
<td>fs</td>
<td>fd</td>
<td>CVT.PS.PW</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>010001</td>
<td>10100</td>
<td>00000</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>100110</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:** CVT.PS.PW fd,fs

**Purpose:**
To convert a pair of 32-bit fixed point words to FP paired-single value

**Description:**
\[ fd \leftarrow (\text{convert\_and\_round}(fs_{63..32}) \ || \ \text{convert\_and\_round}(fs_{31..0}) \) \]

The value in FPR \( fs \), in format \( PW \), is converted to a value in paired-single floating point format and rounded according to the current rounding mode in \( FCSR \). The result is placed in FPR \( fd \).

**Restrictions:**
The fields \( fs \) and \( fd \) must specify valid FPRs---\( fs \) for type \( PW \) and \( fd \) for type \( PS \). If they are not valid, the result is UNPREDICTABLE. The operand in register \( fs \) must be a value in format type \( PW \); if it is not, the result is UNPREDICTABLE and the value in the operand FPR becomes UNPREDICTABLE.

The result of this instruction is UNPREDICTABLE if the processor is executing in 16 FP registers mode.

**Operation:**
\[
\text{StoreFPR}(fd, PS, \\
\quad \text{ConvertFmt}(\text{ValueFPR}(fs, PW)_{63..32}, W, S) \ || \\
\quad \text{ConvertFmt}(\text{ValueFPR}(fs, PW)_{31..0}, W, S))
\]

**Exceptions:**
Coprocessor Unusable, Reserved Instruction

**Floating Point Exceptions:**
Unimplemented Operation
# Chapter 4 The MIPS-3D™ ASE Instruction Set

## Floating Point Reduction Multiply

<table>
<thead>
<tr>
<th>COP1</th>
<th>fmt</th>
<th>ft</th>
<th>fs</th>
<th>fd</th>
<th>MULR.PS</th>
</tr>
</thead>
<tbody>
<tr>
<td>010001</td>
<td>10110</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>

### Format:

MULR.PS fd, fs, ft

### Purpose:

To perform a reduction multiply on two paired-single floating point values

### Description:

\[
\text{fd.PL} \leftarrow \text{ft.PU} \times \text{ft.PL}; \quad \text{fd.PU} \leftarrow \text{fs.PU} \times \text{fs.PL}
\]

The paired-single values in FPR \( ft \) are multiplied together and the result put in the lower paired-single position of FPR \( fd \). Similarly, the paired-single values in FPR \( fs \) are multiplied together and the result put in the upper paired-single position of FPR \( fd \). The two results are calculated to infinite precision and rounded by using the current rounding mode in \( FCSR \). The operands and result are values in format PS.

Any generated exceptions in the two independent adds are OR’ed together. \textit{Cause} bits are ORed into the \textit{Flag} bits if no exception is taken.

### Restrictions:

The fields \( fs, ft, \) and \( fd \) must specify FPRs valid for operands of type PS. If they are not valid, the result is UNPREDICTABLE. The operands must be values in format PS. If they are not, the result is UNPREDICTABLE and the values in the operand FPRs become UNPREDICTABLE.

The result of \texttt{ADDR.PS} is UNPREDICTABLE if the processor is executing in 16 FP registers mode.

### Operation:

\[
\begin{align*}
\text{lower} & \leftarrow \text{ValueFPR}(ft, \text{PS})_{31..0} \times \text{ValueFPR}(ft, \text{PS})_{63..32} \\
\text{upper} & \leftarrow \text{ValueFPR}(fs, \text{PS})_{31..0} \times \text{ValueFPR}(fs, \text{PS})_{63..32} \\
\text{StoreFPR} & (fd, \text{PS}, \text{upper} || \text{lower})
\end{align*}
\]

### Exceptions:

Coprocessor Unusable, Reserved Instruction

### Floating Point Exceptions:

Unimplemented Operation, Invalid Operation, Overflow, Inexact, Underflow
Floating Point Reduced Precision Reciprocal (Sequence Step 1)  

<table>
<thead>
<tr>
<th>COP1</th>
<th>fmt</th>
<th>0</th>
<th>fs</th>
<th>fd</th>
<th>RECIP1</th>
</tr>
</thead>
<tbody>
<tr>
<td>010001</td>
<td>00000</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>011101</td>
</tr>
</tbody>
</table>

**Format:**  
RECIP1.S fd,fs  
RECIP1.D fd,fs  
RECIP1.US fd,fs  
MIPS-3D  
MIPS_3D  
MIPS_3D  

**Purpose:**  
Generate a reduced-precision reciprocal of one or two FP values  

**Description:**  
\[ fd \leftarrow 1.0 / fs \]  
The reciprocal of the value in FPR \( fs \) is approximated and placed in FPR \( fd \). The operand and result are values in format S, D, or PS.  
The numeric accuracy of this operation is implementation dependent; it does not meet the accuracy specified by the IEEE 754 Floating Point standard. A minimum accuracy of 14 bits is recommended for the S input data format, and a minimum accuracy of 23 bits is recommended for the D input data format.  
It is implementation dependent whether the result is affected by the current rounding mode in FCSR. This instruction is meant to operate in RN (round to nearest) mode for the best accuracy. It is also meant to operate in the Flush to Zero (FS=0) mode. In this mode, if the incoming data is in the denormalized range, it is assumed to be zero, and if the output is in the denormalized range, it is forced to zero.  
In addition, if the input to this instruction is zero, the output is not infinity, but the maximum normalized value. This property is useful for 3D graphics applications. If the input is infinity, the output is zero.  
This instruction is used as the first step of an instruction sequence that can be used to produce a full precision reciprocal value. See the description of RECIP2.fmt for an example of how to use this instruction in a code sequence to produce a full precision reciprocal result.  

**Restrictions:**  
The fields \( fs \) and \( fd \) must specify FPRs valid for operands of type \( fmt \). If they are not valid, the result is UNPREDICTABLE. The format of the data in the specified operand register \( fs \) must be a value in format \( fmt \); if it is not, the result is UNPREDICTABLE and the value of the operand FPR becomes UNPREDICTABLE.
Floating Point Reduced Precision Reciprocal (Sequence Step 1, cont.)

**Operation:**

\[ \text{StoreFPR}(fd, fmt, \frac{1.0}{\text{ValueFPR}(fs, fmt)}_{\text{ReducedPrecision}}) \]

**Exceptions:**

Coprocessor Unusable, Reserved Instruction

**Floating Point Exceptions:**

Unimplemented Operation, Invalid Operation, Overflow, Inexact, Underflow, Division-by-zero
### Floating Point Reduced Precision Reciprocal (Sequence Step 2)  
**RECIP2.fmt**

<table>
<thead>
<tr>
<th>COP1</th>
<th>fmt</th>
<th>0</th>
<th>fs</th>
<th>fd</th>
<th>RECIP2</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>010001</td>
<td>00000</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>

**Format:**
- `RECIP2.S fd,fs,ft`
- `RECIP2.D fd,fs,ft`
- `RECIP2.PS fd,fs,ft`

**Purpose:**
Take the result of `RECIP1.fmt` and iterate towards obtaining a full precision reciprocal FP value.

**Description:**
- `fd <- iterate with fs and ft`

This is the second step in the instruction sequence used to generate a full precision reciprocal result. (`RECIP1.fmt` instruction is the first step). The operand and result are values in format S, D, or PS.

The numeric accuracy of this operation is implementation dependent; it does not meet the accuracy specified by the IEEE 754 Floating Point standard.

It is implementation dependent whether the result is affected by the current rounding mode in `FCSR`. This instruction is meant to operate in RN (round to nearest) mode for the best accuracy. It is also meant to operate in the Flush to Zero (FS=0) mode. In this mode, if the incoming data is in the denormalized range, it is assumed to be zero, and if the output is in the denormalized range, it is forced to zero.

The example below shows how a full precision reciprocal result can be obtained using the `RECIP1` and `RECIP2` instructions. Assume that a value `b` is in register `f0` in format S. Assume that `RECIP1.fmt` produces a 16-bit result. At the end of the three-instruction sequence shown below, register `f3` contains the full precision 24-bit reciprocal `1/b`.

```assembly
RECIP1.S f1, f0  /* reduced precision 16-bit 1/b */
RECIP2.S f2, f1, f0 /* -(b * f1 - 1.0) */
MADD.S f3, f1, f1, f2 /* 24-bit 1/b */
```

The instruction sequence to produce a double, 52-bit result is as follows:

```assembly
RECIP1.D f1, f0  /* reduced precision 16-bit 1/b */
RECIP2.D f2, f1, f0 /* -(b * f1 - 1.0) */
MADD.D f3, f1, f1, f2 /* 32-bit 1/b */
RECIP2.D f4, f3, f0 /* -(b * f3 - 1.0) */
MADD.D f5, f3, f3, f4 /* 53-bit 1/b */
```

The instruction sequence to take a paired single value and produce a paired single result is as follows. Assume that register `f0` holds two single values `a` and `b` in a paired single format, i.e., `f0 ← [a | b]`.

```assembly
RECIP1.PS f1, f0  /* ( reduced precision 16-bit 1/a and 1/b ) */
RECIP2.PS f2, f1, f0 /* ( -(a*f1-1.0) and -(b*f1-1.0) ) */
MADD.PS f3, f1, f1, f2 /* ( 24-bit 1/a and 1/b ) */
```
If the hardware does not implement the RECIP1.PS instruction, it is still possible to obtain a paired single result, but this takes three more instructions in the required sequence. Assume that register f0 holds a single value a and register f1 holds a single value b.

```
RECP1.S f2, f0  /* ( f2 gets reduced precision 1/a ) */
RECP1.S f3, f1  /* ( f3 gets reduced precision 1/b ) */
CVT.PS.S f4, f1, f0 /* ( f4 now holds the PS values b | a ) */
CVT.PS.S f5, f3, f2 /* ( f5 holds PS seed 1/b | 1/a ) */
RECP2.PS f6, f5, f4 /* ( f6 holds intermediate 1/b | 1/a ) */
MADD.PS f7, f5, f5, f6 /* ( f7 holds full precision PS 1/b | 1/a ) */
```

**Restrictions:**

The fields fs, ft, and fd must specify FPRs valid for operands of type fmt. If they are not valid, the result is UNPREDICTABLE. The format of the data in the specified operand register fs must be a value in format fmt; if it is not, the result is UNPREDICTABLE and the value in the operand FPR becomes UNPREDICTABLE.

The result of RECIP2.PS is UNPREDICTABLE if the processor is executing in 16 FP registers mode.

**Operation:**

```
StoreFPR(fd, fmt, RECIP_iteration(ValueFPR(fs, fmt), ValueFPR(ft, fmt))
```

**Exceptions:**

Coprocessor Unusable, Reserved Instruction

**Floating Point Exceptions:**

Unimplemented Operation, Inexact, Invalid Operation, Overflow, Underflow
Floating Point Reduced Precision Reciprocal Square Root (Sequence Step 1)  

RSQRT1.fmt

<table>
<thead>
<tr>
<th>COP1</th>
<th>fmt</th>
<th>0</th>
<th>fs</th>
<th>fd</th>
<th>RSQRT1</th>
</tr>
</thead>
<tbody>
<tr>
<td>010001</td>
<td>0</td>
<td>00000</td>
<td>5</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>

**Format:**  
- RSQRT1.S  
- RSQRT1.D  
- RSQRT1.PS

**Purpose:**  
To produce a reduced-precision reciprocal of the square root of one or two FP values

**Description:**  
\[ fd \leftarrow \frac{1.0}{\sqrt{fs}} \]

The reciprocal of the positive square root of the value in FPR fs is approximated and placed in FPR fd. The operand and result are values in format S, D, or PS.

The numeric accuracy of this operation is implementation dependent; it does not meet the accuracy specified by the IEEE 754 Floating Point standard. A minimum accuracy of 14 bits is recommended for the S input data format, and 23 bits for the D data format.

It is implementation dependent whether the result is affected by the current rounding mode in FCSR.

In addition, if the input to this instruction is zero, the output is not infinity, but the maximum normalized value. This property is useful for 3D graphics applications. If the input is infinity, the output is zero.

This instruction is used as the first step of an instruction sequence that can be used to produce a full precision reciprocal square root value. See the description of RSQRT2.fmt for an example of how to use this instruction in a code sequence to produce a full precision reciprocal square root result.

**Restrictions:**  
The fields fs and fd must specify FPRs valid for operands of type fmt. If they are not valid, the result is UNPREDICTABLE. The format of the data in the specified operand register fs must be a value in format fmt; if it is not, the result is UNPREDICTABLE and the value in the operand FPR becomes UNPREDICTABLE.

**Operation:**  
\[ \text{StoreFPR}(fd, \text{fmt}, (1.0 / \text{SquareRoot}(\text{ValueFPR}(fs, \text{fmt}))))_{\text{ReducedPrecision}} \]
Floating Point Reduced Precision Reciprocal Square Root (Sequence Step 1, cont.)  RSQRT1.fmt

**Exceptions:**
Coprocessor Unusable, Reserved Instruction

**Floating Point Exceptions:**
Unimplemented Operation, Invalid Operation, Overflow, Inexact, Underflow, Division-by-zero
Floating Point Reduced Precision Reciprocal Square Root (Sequence Step 2)  

<table>
<thead>
<tr>
<th>COP1</th>
<th>fmt</th>
<th>0</th>
<th>fs</th>
<th>fd</th>
<th>RSQRT2</th>
</tr>
</thead>
<tbody>
<tr>
<td>010001</td>
<td>00000</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>

**Format:**
- `RSQRT2.S  fd, fs, ft`  
- `RSQRT2.D  fd, fs, ft`  
- `RSQRT2.PS fd, fs, ft`  

**Purpose:**
Iterate towards obtaining a full precision reciprocal square root FP value

**Description:**
- `fd <- iterate with fs and ft`

This is a step of iteration towards generating the full precision reciprocal square root value. The operand and result are values in format S, D, or PS.

The numeric accuracy of this operation is implementation dependent; it does not meet the accuracy specified by the IEEE 754 Floating Point standard.

It is implementation dependent whether the result is affected by the current rounding mode in FCSR.

A full precision reciprocal square root result is obtained by using the instruction sequence shown below. Assume that a value `b` is in register `f0` in format S. Assume that `RSQRT1.fmt` has a 16-bit precision in the example implementation. At the end of the four-instruction sequence shown below, register `f4` contains the full precision 24-bit reciprocal square root `1/(sqrt)b`.

- `RSQRT1.S f1, f0 /* 16-bit 1/sqrt(b) */`
- `MUL.S f2, f1, f0 /* b * f0 */`
- `RSQRT2.S f3, f2, f1 /* -(f1 * f2 - 1.0)/2 */`
- `MADD.S f4, f1, f1, f3 /* 24-bit 1/sqrt(b) */`

The instruction sequence to produce a 52-bit result is as follows:

- `RSQRT1.D f1, f0 /* 16-bit 1/sqrt(b) */`
- `MUL.D f2, f1, f0 /* b * f0 */`
- `RSQRT2.D f3, f2, f1 /* -(f1 * f2 - 1.0)/2 */`
- `MADD.D f4, f1, f1, f3 /* 31-bit 1/sqrt(b) */`
- `MUL.D f5, f0, f4 /* b * f0 */`
- `RSQRT2.D f6, f5, f4 /* -(f4 * f5 - 1.0)/2 */`
- `MADD.D f7, f4, f4, f6 /* 53-bit 1/sqrt(b) */`

The instruction sequence to take a paired single value and produce a paired single result is as follows. Assume that register `f0` holds two single values `a` and `b` in a paired single format, i.e., `f0 ← a | b`.

- `RSQRT1.PS f1, f0 /* ( 16-bit 1/sqrt(a) and 1/sqrt(b) ) */`
- `MUL.PS f2, f1, f0 /* ( a * f0 and b * f1 ) */`
- `RSQRT2.PS f3, f2, f1 /* ( -(f1*f2-1-0)/2 ) */`
- `MADD.PS f4, f1, f1, f3 /* ( 24-bit 1/sqrt(a) and 1/sqrt(b) ) */`
If the hardware does not implement the RSQRT1.PS instruction, it is still possible to obtain a paired single result, but this takes three more instructions in the required sequence. Assume that register f0 holds a single value a and register f1 holds a single value b.

```assembly
RSQRT1.S f2, f0  /* ( f2 gets reduced precision 1/sqrt(a) ) */
RSQRT1.S f3, f1  /* ( f3 gets reduced precision 1/sqrt(b) ) */
CVT.PS.S f4, f1, f0  /* ( f4 now holds the PS values b | a ) */
CVT.PS.S f5, f3, f2  /* ( f5 holds PS seed 1/sqrt(b) | 1/sqrt(a) ) */
MUL.PS f6, f5, f4  /* ( f6 holds intermediate1 results ) */
RSQRT2.PS f7, f6, f5  /* ( f7 holds intermediate2 results ) */
MADD.PS f8, f5, f5, f7  /* ( f8 holds full precision PS 1/sqrt(b) | */
  /* 1/sqrt(a) ) */
```

**Restrictions:**

The fields fs, ft, and fd must specify FPRs valid for operands of type fmt. If they are not valid, the result is UNPREDICTABLE. The format of the data in the specified operand register fs must be a value in format fmt; if it is not, the result is UNPREDICTABLE and the value of the operand FPR becomes UNPREDICTABLE.

**Operation:**

```assembly
StoreFPR(fd, fmt, RSQRT_iteration(ValueFPR(fs, fmt), ValueFPR(ft, fmt)))
```

**Exceptions:**

Coprocessor Unusable, Reserved Instruction

**Floating Point Exceptions:**

Unimplemented Operation, Invalid Operation, Overflow, Inexact, Underflow
### Revision History

<table>
<thead>
<tr>
<th>Revision</th>
<th>Date</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.00</td>
<td>August 6, 1999</td>
<td>First external release</td>
</tr>
<tr>
<td>1.10</td>
<td>November 1, 2000</td>
<td>Convert format and include document in document set</td>
</tr>
<tr>
<td>1.11</td>
<td>March 12, 2001</td>
<td>Add architecture requirements and subsetting rules for next external review release.</td>
</tr>
</tbody>
</table>