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

Document Number: MD00099
Revision 2.50
July 1, 2005

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

Template: B1.14, Built with tags: 2B ARCH FPU_PS FPU_PSandARCH MIPS64

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
1.2.3 UNSTABLE 2
1.3 Special Symbols in Pseudocode Notation 3
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 22
2.4 FPU Instructions 23

Chapter 3 MIPS-3D® Application-Specific Extension to the MIPS64® Architecture 25
3.1 Base Architecture Requirements 25
3.2 Software Detection of the ASE 25
3.3 Compliance and Subsetting 25
3.4 MIPS-3D Overview 25
3.5 Instruction Bit Encoding 26

Chapter 4 The MIPS-3D® ASE Instruction Set 29
4.1 MIPS-3D Instruction Descriptions 29

Appendix A Revision History 53

List of Figures

Figure 2-1: Example of Instruction Description 8
Figure 2-2: Example of Instruction Fields 9
Figure 2-3: Example of Instruction Descriptive Name and Mnemonic 9
Figure 2-4: Example of Instruction Format 9
Figure 2-5: Example of Instruction Purpose 10
Figure 2-6: Example of Instruction Description 10
Figure 2-7: Example of Instruction Restrictions 11
Figure 2-8: Example of Instruction Operation 11
Figure 2-9: Example of Instruction Exception 11
Figure 2-10: Example of Instruction Programming Notes 12
Figure 2-11: COP_LW Pseudocode Function 13
Figure 2-12: COP_LD Pseudocode Function 13
Figure 2-13: COP_SW Pseudocode Function 13
Figure 2-14: COP_SD Pseudocode Function 14
Figure 2-15: CoprocessorOperation Pseudocode Function 14
Figure 2-16: AddressTranslation Pseudocode Function 15
Figure 2-17: LoadMemory Pseudocode Function 15
Figure 2-18: StoreMemory Pseudocode Function 16
Figure 2-19: Prefetch Pseudocode Function 16
Figure 2-20: SyncOperation Pseudocode Function 17
Figure 2-21: ValueFPR Pseudocode Function 18
Figure 2-22: StoreFPR Pseudocode Function 19
Figure 2-23: CheckFPException Pseudocode Function 20
Figure 2-24: FPConditionCode Pseudocode Function 20
Figure 2-25: SetFPConditionCode Pseudocode Function 20
Figure 2-26: SignalException Pseudocode Function 21
Figure 2-27: SignalDebugBreakpointException Pseudocode Function 21
Figure 2-28: SignalDebugModeBreakpointException Pseudocode Function 21
Figure 2-29: NullifyCurrentInstruction PseudoCode Function 21
Figure 2-30: JumpDelaySlot Pseudocode Function 22
Figure 2-31: NotWordValue Pseudocode Function 22
Figure 2-32: PolyMult Pseudocode Function 22

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-3D® ASE 26
Table 3-2: Symbols Used in the Instruction Encoding Tables 26
Table 3-3: MIPS-3D COP1 Encoding of rs Field 27
Table 3-4: MIPS-3D COP1 Encoding of Function Field When rs=S 27
Table 3-5: MIPS-3D COP1 Encoding of Function Field When rs=D 27
Table 3-6: MIPS-3D COP1 Encoding of Function Field When rs=W or L 27
Table 3-7: MIPS-3D COP1 Encoding of Function Field When rs=PS 28

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 MIPS16e™ 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

1.2.3 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
*, ×	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
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, 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*

Table 1-1 Symbols Used in Instruction Operation Statements

Symbol	Meaning
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).
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:, **I+n**:, **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**. 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. 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, the *ISA Mode* is a single-bit register that determines in which mode the processor is executing, as follows: **Encoding** / **Meaning** 0 — The processor is executing 32-bit MIPS instructions 1 — The processor is executing MIIPS16e instructions 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.

Table 1-1 Symbols Used in Instruction Operation Statements

Symbol	Meaning
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.
FP32RegistersMode	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. 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.
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

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

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

or via E-mail to architecture@mips.com.

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

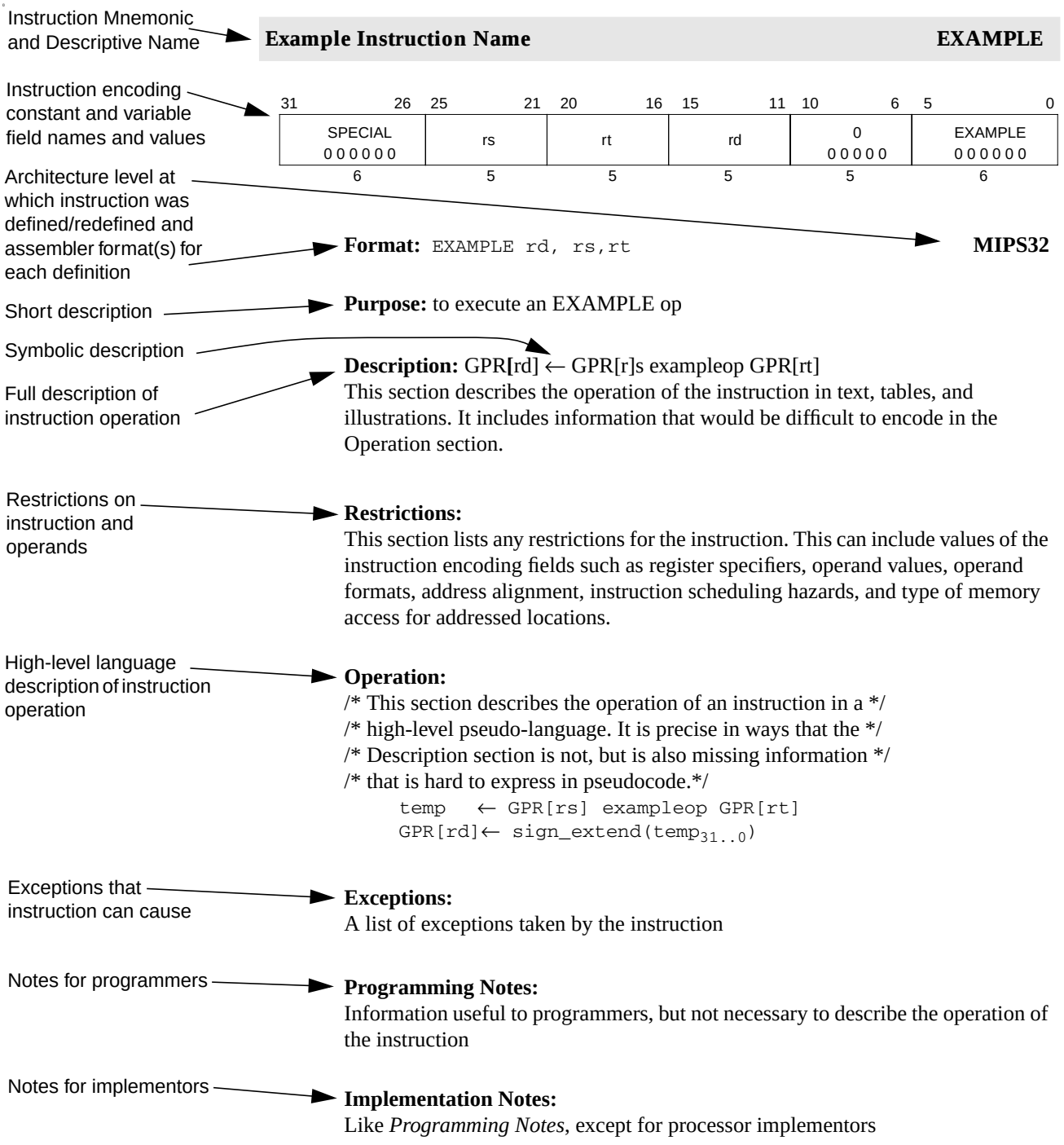


Figure 2-1 Example of Instruction Description

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**.

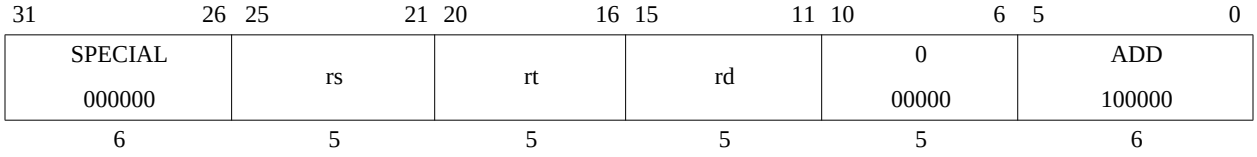


Figure 2-2 Example of Instruction Fields

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.

Add Word **ADD**

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.

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

Figure 2-4 Example of Instruction Format

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.

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: `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*

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:

```
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
```

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

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
- "Memory Operation Functions" on page 14
- "Floating Point Functions" on page 17
- "Miscellaneous Functions" on page 20

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

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 2-15 CoprocessorOperation Pseudocode Function

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 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 2-16 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.

```
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:      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 2-17 LoadMemory Pseudocode Function

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.

```
StoreMemory (CCA, AccessLength, MemElem, pAddr, 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 2-18 StoreMemory Pseudocode Function

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 2-19 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

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.

```
SyncOperation(stype)

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

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

endfunction SyncOperation
```

Figure 2-20 SyncOperation Pseudocode Function

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.

```
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 (fpr0 ≠ 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
```

Figure 2-21 ValueFPR Pseudocode Function

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

```
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
```

Figure 2-22 StoreFPR Pseudocode Function

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

CheckFPException

```
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
```

Figure 2-23 CheckFPException Pseudocode Function

FPConditionCode

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

```
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
```

Figure 2-24 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 ← FCSR31..24 || tf || FCSR22..0
   else
      FCSR ← FCSR31..25+cc || tf || FCSR23+cc..0
endif

endfunction SetFPConditionCode
```

Figure 2-25 SetFPConditionCode Pseudocode Function

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.

```
SignalException(Exception, argument)

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

endfunction SignalException
```

Figure 2-26 SignalException Pseudocode Function

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.

```
SignalDebugBreakpointException()

endfunction SignalDebugBreakpointException
```

Figure 2-27 SignalDebugBreakpointException Pseudocode Function

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.

```
SignalDebugModeBreakpointException()

endfunction SignalDebugModeBreakpointException
```

Figure 2-28 SignalDebugModeBreakpointException Pseudocode Function

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.

```
NullifyCurrentInstruction()

endfunction NullifyCurrentInstruction
```

Figure 2-29 NullifyCurrentInstruction PseudoCode Function

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.

```
JumpDelaySlot(vAddr)

   /* vAddr:Virtual address */

endfunction JumpDelaySlot
```

Figure 2-30 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.

```
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
```

Figure 2-31 NotWordValue Pseudocode Function

PolyMult

The PolyMult function multiplies two binary polynomial coefficients.

```
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
```

Figure 2-32 PolyMult Pseudocode Function

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 Section 2.3, "Op and Function Subfield Notation" on page 22 for a description of the *op* and *function* subfields.

MIPS-3D® Application-Specific Extension to the MIPS64® Architecture

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:

- **A 64-bit floating point unit 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.

In Release 1 of the Architecture, the MIPS-3D ASE was supported only on MIPS64 implementations. In Release 2 of the Architecture, MIPS-3D is supported with a 64-bit floating point unit (as denoted by FIR_{F64}), whether on a MIPS32 or MIPS64 processor.

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 Compliance and Subsetting

There are no instruction subsets of the MIPS-3D ASE — all MIPS-3D instructions and data types must be implemented.

3.4 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 29, 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)

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

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

3.5 Instruction Bit Encoding

Table 3-3 through Table 3-7 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 3-2 Symbols Used in the Instruction Encoding Tables

Symbol	Meaning
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.
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.
∇	Operation or field codes marked with this symbol represent instructions which were only legal if 64-bit operations were enabled on implementations of Release 1 of the Architecture. In Release 2 of the architecture, operation or field codes marked with this symbol represent instructions which are legal if 64-bit floating point operations are enabled. In other cases, executing such an instruction must cause a Reserved Instruction Exception (non-coprocessor encodings or coprocessor instruction encodings for a coprocessor to which access is allowed) or a Coprocessor Unusable Exception (coprocessor instruction encodings for a coprocessor to which access is not allowed).

Table 3-3 MIPS-3D *COP1* Encoding of rs Field

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		*BC1ANY2* δεV	*BC1ANY4* δεV					
2	10								
3	11								

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

function		*bits 2..0*							
		0	1	2	3	4	5	6	7
bits 5..3		000	001	010	011	100	101	110	111
0	000								
1	001								
2	010								
3	011					RECIP2 εV	RECIP1 εV	RSQRT1 εV	RSQRT2 εV
4	100								
5	101								
6	110	CABS.F εV	CABS.UN εV	CABS.EQ εV	CABS.UEQ εV	CABS.OLT εV	CABS.ULT εV	CABS.OLE εV	CABS.ULE εV
7	111	CABS.SF εV	CABS.NGLE εV	CABS.SEQ εV	CABS.NGL εV	CABS.LT εV	CABS.NGE εV	CABS.LE εV	CABS.NGT εV

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

function		*bits 2..0*							
		0	1	2	3	4	5	6	7
bits 5..3		000	001	010	011	100	101	110	111
0	000								
1	001								
2	010								
3	011					RECIP2 εV	RECIP1 εV	RSQRT1 εV	RSQRT2 εV
4	100								
5	101								
6	110	CABS.F εV	CABS.UN εV	CABS.EQ εV	CABS.UEQ εV	CABS.OLT εV	CABS.ULT εV	CABS.OLE εV	CABS.ULE εV
7	111	CABS.SF εV	CABS.NGLE εV	CABS.SEQ εV	CABS.NGL εV	CABS.LT εV	CABS.NGE εV	CABS.LE εV	CABS.NGT εV

Table 3-6 MIPS-3D *COP1* Encoding of Function Field When rs=*W or L*

function		*bits 2..0*							
		0	1	2	3	4	5	6	7
bits 5..3		000	001	010	011	100	101	110	111
0	000								
1	001								
2	010								
3	011								
4	100							CVT.PS.PW εV	
5	101								
6	110								
7	111								

Table 3-7 MIPS-3D *COP1* Encoding of Function Field When rs=*PS*

function	*bits 2..0*							
	0	1	2	3	4	5	6	7
bits 5..3	000	001	010	011	100	101	110	111
0 000								
1 001								
2 010								
3 011	ADDR εV		MULR εV		RECIP2 εV	RECIP1 εV	RSQRT1 εV	RSQRT2 εV
4 100					CVT.PW.PS εV			
5 101					PLL.PS εV	PLU.PS εV	PUL.PS εV	PUU.PS εV
6 110	CABS.F εV	CABS.UN εV	CABS.EQ εV	CABS.UEQ εV	CABS.OLT εV	CABS.ULT εV	CABS.OLE εV	CABS.ULE εV
7 111	CABS.SF εV	CABS.NGLE εV	CABS.SEQ εV	CABS.NGL εV	CABS.LT εV	CABS.NGE εV	CABS.LE εV	CABS.NGT εV

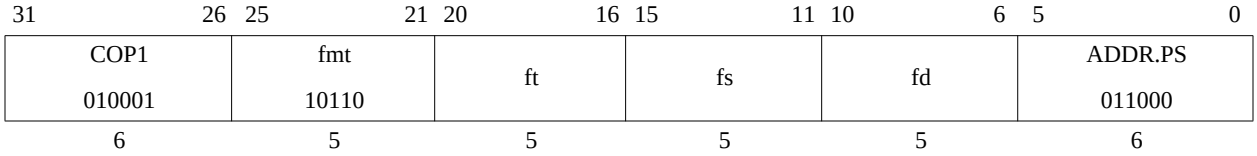
±

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 — ADDR.fmt



Format: `ADDR.PS fd, fs, ft` **MIPS-3D**

Purpose:

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

Description: `FPR[fd].PL ← FPR[ft].PU + FPR[ft].PL; FPR[fd].PU ← FPR[fs].PU + FPR[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 ← ValueFPR(ft, PS)31..0 + ValueFPR(ft, PS)63..32
upper ← ValueFPR(fs, PS)31..0 + ValueFPR(fs, PS)63..32
StoreFPR (fd, PS, upper || lower)
```

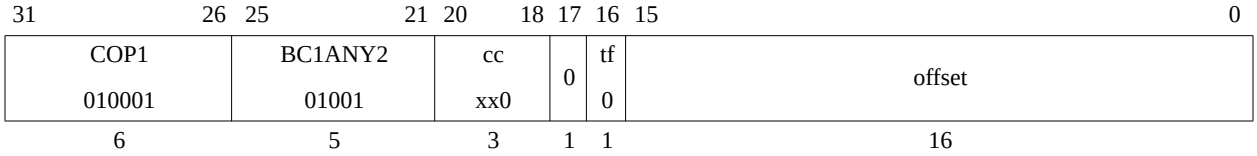
Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

Unimplemented Operation, Invalid Operation, Overflow, Inexact, Underflow

Branch on Any of Two Floating Point Condition Codes False — BC1ANY2F



Format: `BC1ANY2F cc,offset` **MIPS-3D**

Purpose:

To test two consecutive floating point condition codes and do a PC-relative conditonal branch

Description: `If FPConditionCode(CCn+1) = 0 or FPConditionCode(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 CC_5 or CC_4 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 || 02
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

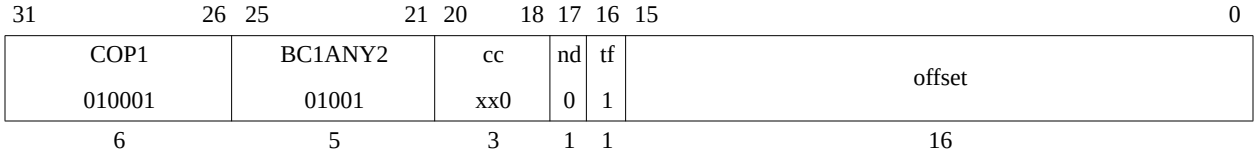
Floating Point Exceptions:

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 — BC1ANY2T



Format: `BC1ANY2T cc,offset` **MIPS-3D**

Purpose:

To test two consecutive FP condition codes and do a PC-relative conditonal branch

Description: `If FPConditionCode(CCn+1) = 1 or FPConditionCode(CCn) = 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*.

```
I:      condition ← (FPConditionCode(cc)   = 1) or
                    (FPConditionCode(cc+1) = 1)
        target_offset ← (offset15)GPRLEN-(16+2) || offset || 02
I+1:    if condition then
           PC ← PC + target_offset
        endif
```

Branch on Any of Two Floating Point Condition Codes True, cont. **BC1ANY2T**

Exceptions:

Coprocessor Unusable, Reserved Instruction

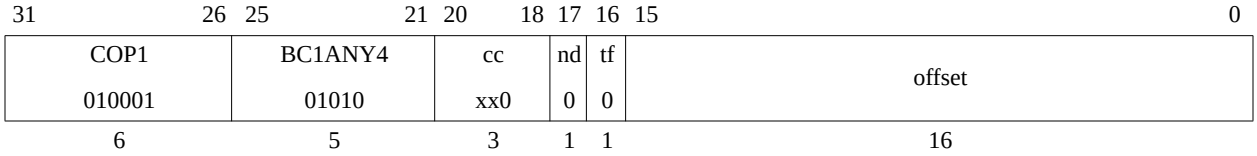
Floating Point Exceptions:

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 False — BC1ANY4F



Format: `BC1ANY4F cc,offset` **MIPS-3D**

Purpose:

To test four consecutive FP condition codes and do a PC-relative conditonal branch

Description: `If FPConditionCode(CCn+3) = 0 or FPConditionCode(CCn+2) = 0 or FPCondition-Code(CCn+1) = 0 or FPConditionCode(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 $CC_{3..0}$ 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:      condition ← (FPConditionCode(cc)   = 0) or
                    (FPConditionCode(cc+1) = 0) or
                    (FPConditionCode(cc+2) = 0) or
                    (FPConditionCode(cc+3) = 0)
        target_offset ← (offset15)GPRLEN-(16+2) || offset || 02
I+1:    if condition then
           PC ← PC + target_offset
        endif
```

Branch on Any of Four Floating Point Condition Codes False, cont. **BC1ANY4F**

Exceptions:

Coprocessor Unusable, Reserved Instruction

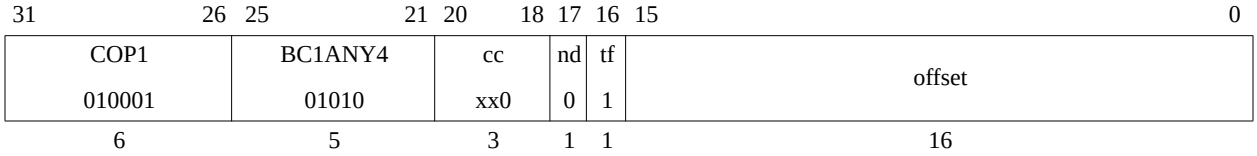
Floating Point Exceptions:

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 — BC1ANY4T



Format: `BC1ANY4T cc,offset` **MIPS-3D**

Purpose:

To test four consecutive FP condition codes and do a PC-relative conditonal branch

Description: `If FPConditionCode(CCn+3) = 1 or FPConditionCode(CCn+2) = 1 or FPConditionCode(CCn+1) = 1 or FPConditionCode(CCn) = 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 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 **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:      condition ← (FPConditionCode(cc)   = 1) or
                    (FPConditionCode(cc+1) = 1) or
                    (FPConditionCode(cc+2) = 1) or
                    (FPConditionCode(cc+3) = 1)
        target_offset ← (offset15)GPRLEN-(16+2) || offset || 02
I+1:    if condition then
           PC ← PC + target_offset
        endif
```

Branch on Any of Four Floating Point Condition Codes True, cont. **BC1ANY4T**

Exceptions:

Coprocessor Unusable, Reserved Instruction

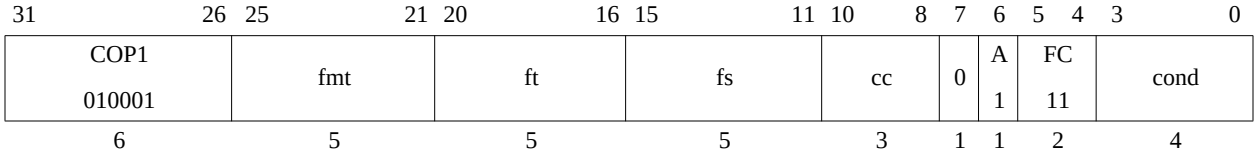
Floating Point Exceptions:

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 — CABS.cond.fmt



Format: `CABS.cond.S cc,fs,ft` **MIPS-3D**

`CABS.cond.D cc,fs,ft` **MIPS-3D**

`CABS.cond.PS cc,fs,ft` **MIPS-3D**

Purpose:

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

Description: `FPConditionCode(cc) ← FPR[fs]` *compare_absolute_cond* `FPR[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 $cond_{2..1}$ 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.

Floating Point Absolute Compare, cont. **CABS.cond.fmt**

Operation:

```
if SNaN(ValueFPR(fs, fmt)) or SNaN(ValueFPR(ft, fmt)) or
   QNaN(ValueFPR(fs, fmt)) or QNaN(ValueFPR(ft, fmt)) then
   less ← false
   equal ← false
   unordered ← true
   if (SNaN(ValueFPR(fs,fmt)) or SNaN(ValueFPR(ft,fmt))) or
   (cond3 and (QNaN(ValueFPR(fs,fmt)) or QNaN(ValueFPR(ft,fmt)))) then
      SignalException(InvalidOperation)
   endif
else
   less ←    AbsoluteValue(ValueFPR(fs, fmt)) <fmt
             AbsoluteValue(ValueFPR(ft, fmt))
   equal ←   AbsoluteValue(ValueFPR(fs, fmt)) =fmt
             AbsoluteValue(ValueFPR(ft, fmt))
   unordered ← false
endif
condition ← (cond2 and less) or (cond1 and equal)
      or (cond0 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.

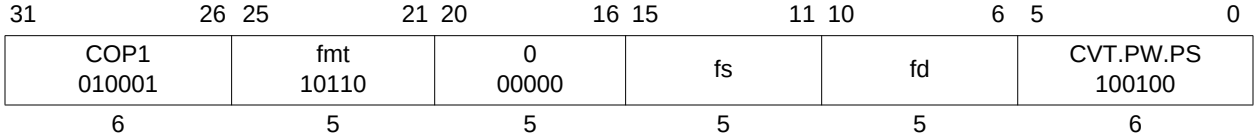
Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

Unimplemented Operation, Invalid Operation

Floating Point Convert Paired Single to Paired Word — CVT.PW.PS



Format: `CVT.PW.PS fd,fs` **MIPS-3D**

Purpose:

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

Description: `FPR[fd].PU ← convert_and_round(FPR[fs].PU); FPR[fd].PL ← convert_and_round(FPR[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:

```
StoreFPR(fd, PW,
                ConvertFmt(ValueFPR(fs, PS)63..32, S, W) ||
                ConvertFmt(ValueFPR(fs, PS)31..0, S, W)
        )
```

Floating Point Convert Paired Single to Paired Word (cont.) **CVT.PW.PS**

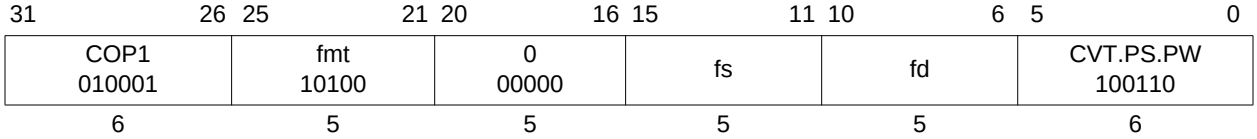
Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions

Unimplemented Operation, Invalid Operation, Overflow, Inexact

Floating Point Convert Paired Word to Paired Single — CVT.PS.PW



Format: `CVT.PS.PW fd,fs` **MIPS-3D**

Purpose:

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

Description: $FPR[fd] \leftarrow (convert\_and\_round(FPR[fs]_{63..32}) \parallel convert\_and\_round(FPR[f]s_{31..0})$

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:

```
StoreFPR(fd, PS,
                ConvertFmt(ValueFPR(fs, PW)63..32, W, S) ||
                ConvertFmt(ValueFPR(fs, PW)31..0, W, S)
        )
```

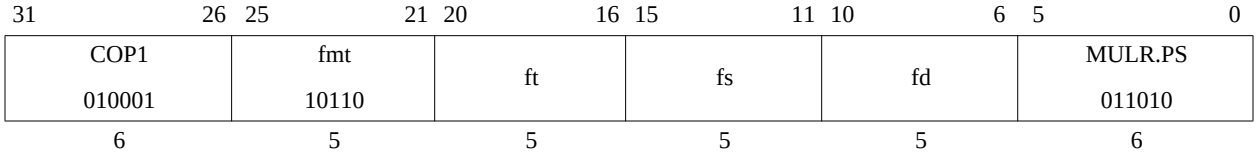
Exceptions:

Coprocessor Unusable, Reserved Instruction

***Floating Point Exceptions*:**

Unimplemented Operation

Floating Point Reduction Multiply — MULR.PS



Format: `MULR.PS fd, fs, ft` **MIPS-3D**

Purpose:

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

Description: `FPR[fd].PL ← FPR[ft].PU * FPR[ft].PL; FPR[fd].PU ← FPR[fs].PU * FPR[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. *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 ← ValueFPR(ft, PS)31..0 × ValueFPR(ft, PS)63..32
upper ← ValueFPR(fs, PS)31..0 × ValueFPR(fs, PS)63..32
StoreFPR (fd, PS, upper || lower)
```

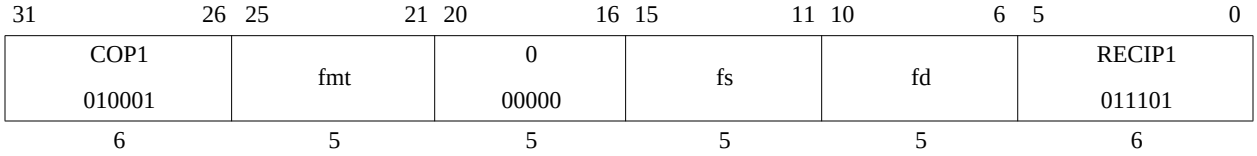
Exceptions:

Coprocessor Unusable, Reserved Instruction

Floating Point Exceptions:

Unimplemented Operation, Invalid Operation, Overflow, Inexact, Underflow

Floating Point Reduced Precision Reciprocal (Sequence Step 1) RECIP1.fmt



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

`RECIP1.D fd,fs` **MIPS_3D**

`RECIP1.PS fd,fs` **MIPS_3D**

Purpose:

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

Description: `FPR[fd] ← 1.0 / FPR[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 both the S and D input data formats.

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.) **RECIP1.fmt**

Operation:

```
StoreFPR(fd, fmt, (1.0 / ValueFPR(fs, fmt))ReducedPrecision)
```

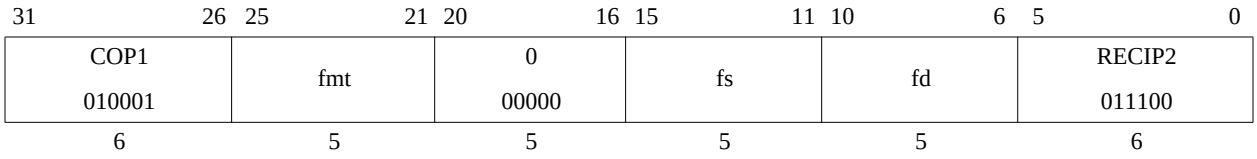
Exceptions:

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



Format: `RECIP2.S fd,fs,ft` **MIPS-3D**

`RECIP2.D fd,fs,ft` **MIPS-3D**

`RECIP2.PS fd,fs,ft` **MIPS-3D**

Purpose:

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

Description: `FPR[fd] ← iterate with FPR[fs] and FPR[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.

```
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:

```
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.

```
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 ) */
```

Floating Point Reduced Precision Reciprocal (Sequence Step 2, cont.) RECIP2.fmt

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.

```
RECIP1.S  f2, f0          /* ( f2 gets reduced precision 1/a ) */
RECIP1.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 ) */
RECIP2.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)))
```

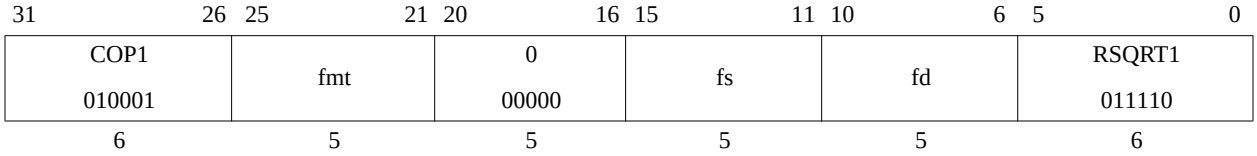
Exceptions:

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



Format:

```
RSQRT1.S    fd, fs                    MIPS-3D
RSQRT1.D    fd, fs                    MIPS-3D
RSQRT1.PS   fd, fs                    MIPS-3D
```

Purpose:

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

Description: `FPR[fd] ← 1.0 / sqrt (FPR[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:

```
StoreFPR(fd, fmt, (1.0 / SquareRoot(ValueFPR(fs, fmt)))ReducedPrecision)
```

Floating Point Reduced Precision Reciprocal Square Root (Sequence Step 1, cont.) **RSQRT1.fmt**

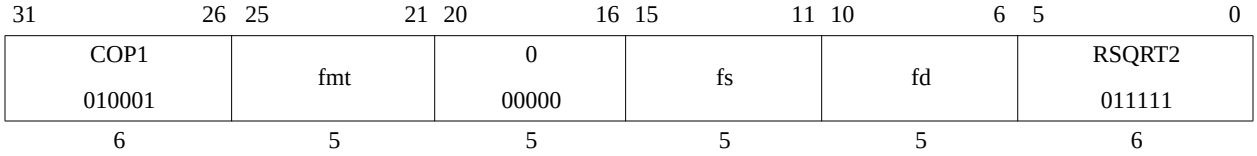
Exceptions:

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) RSQRT2.fmt



Format: `RSQRT2.S fd, fs, ft` **MIPS-3D**
`RSQRT2.D fd, fs, ft` **MIPS-3D**
`RSQRT2.PS fd, fs, ft` **MIPS-3D**

Purpose:

Iterate towards obtaining a full precision reciprocal square root FP value

Description: `FPR[fd] ← iterate with FPR[fs] and FPR[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) ) */
```

Floating Point Reduced Precision Reciprocal Square Root (Sequence Step 2, cont.) RSQRT2.fmt

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.

```
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:

```
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

In the left hand page margins of this document you may find vertical change bars to note the location of significant changes to this document since its last release. Significant changes are defined as those which you should take note of as you use the MIPS IP. Changes to correct grammar, spelling errors or similar may or may not be noted with change bars. Change bars will be removed for changes which are more than one revision old.

Please note: Limitations on the authoring tools make it difficult to place change bars on changes to figures. Change bars on figure titles are used to denote a potential change in the figure itself.

Revision	Date	Description
1.00	August 6, 1999	First external release
1.10	November 1, 2000	Convert format and include document in document set
1.11	March 12, 2001	Add architecture requirements and subsetting rules for next external review release.
1.12	August 29, 2002	Update template to synchronize with latest documentation set release.
2.00	May 15, 2003	Changes in this revision: • Update instruction descriptions to allow MIPS-3D to be implemented on a 64-bit FPU (as denoted by FIR_{F64}), whether on a MIPS32 or MIPS64 processor. This reflects changes introduced with Release 2 of the MIPS Architecture.
2.50	July 1, 2005	Changes in this revision: • Modify the recommendation for minimum bits of accuracy in the RECIP1.D instruction from 23 to 14 bits. • Update to FrameMaker 7.1 • Correct copyright year in Architecture for Programmers version