MIPS32™ Architecture For Programmers Volume I: Introduction to the MIPS32™ Architecture

Document Number: MD00082
Revision 0.95
March 12, 2001

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

Copyright © 2001 MIPS Technologies, Inc. All rights reserved.

Unpublished rights reserved under the Copyright Laws of the United States of America.

This document contains information that is proprietary to MIPS Technologies, Inc. (“MIPS Technologies”). Any copying, modifyingor use of this information (in whole or in part) which is not expressly permitted in writing by MIPS Technologies or a contractually-authorized third party is strictly prohibited. At a minimum, this information is protected under unfair competition laws and the expression of the information contained herein is protected under federal copyright laws. Violations thereof may result in criminal penalties and fines.

MIPS Technologies or any contractually-authorized third party reserves the right to change the information contained in this document to improve function, design or otherwise. MIPS Technologies does not assume any liability arising out of the application or use of this information. Any license under patent rights or any other intellectual property rights owned by MIPS Technologies or third parties shall be conveyed by MIPS Technologies or any contractually-authorized third party in a separate license agreement between the parties.

The information contained in this document constitutes one or more of the following: commercial computer software, commercial computer software documentation or other commercial items. If the user of this information, or any related documentation of any kind, including related technical data or manuals, is an agency, department, or other entity of the United States government (“Government”), the use, duplication, reproduction, release, modification, disclosure, or transfer of this information, or any related documentation of any kind, is restricted in accordance with Federal Acquisition Regulation 12.212 for civilian agencies and Defense Federal Acquisition Regulation Supplement 227.7202 for military agencies. The use of this information by the Government is further restricted in accordance with the terms of the license agreement(s) and/or applicable contract terms and conditions covering this information from MIPS Technologies or any contractually-authorized third party.

MIPS, R3000, R4000, R5000, R8000 and R10000 are among the registered trademarks of MIPS Technologies, Inc., and R4300, R20K, MIPS16, MIPS32, MIPS64, MIPS-3D, MIPS I, MIPS II, MIPS III, MIPS IV, MIPS V, MDMX, SmartMIPS, 4K, 4Kc, 4Km, 4Kp, 5K, 5Kc, 20K, 20Kc, EC, MGB, SOC-it, SEAD, YAMON, ATLAS, JALGO, CoreLV and MIPS-based are among the trademarks of MIPS Technologies, Inc.

All other trademarks referred to herein are the property of their respective owners.

Table of Contents

List of Figures

List of Tables

Chapter 1

About This Book

The MIPS32™ Architecture For Programmers Volume I comes as a multi-volume set.

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

1.1 Typographical Conventions

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

1.1.1 Italic Text

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

1.1.2 Bold Text

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

1.1.3 Courier Text

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

1.2 UNPREDICTABLE and UNDEFINED

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

1.2.1 UNPREDICTABLE

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

UNPREDICTABLE results or operations have several implementation restrictions:

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

1.2.2 UNDEFINED

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

UNDEFINED operations or behavior has one implementation restriction:

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

1.3 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
$\leftarrow$	Assignment
$=, \neq$	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.
$x_{y..z}$	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
$*, \times$	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
$\leq$	2's complement less-than or equal comparison
$\geq$	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.
FPR[x]	Floating Point operand register *x*
FCC[CC]	Floating Point condition code CC. *FCC[0]* has the same value as *COC[1]*.
FPR[x]	Floating Point (Coprocessor unit 1), general register *x*
CPR[z,x,s]	Coprocessor unit *z*, general register *x*, select *s*
CCR[z,x]	Coprocessor unit *z*, control register *x*
COC[z]	Coprocessor unit *z* condition signal
Xlat[x]	Translation of the MIPS16 GPR number *x* into the corresponding 32-bit GPR number
BigEndianMem	Endian mode as configured at chip reset (0 $\rightarrow$Little-Endian, 1 $\rightarrow$ 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.

Table 1-1 Symbols Used in Instruction Operation Statements

Symbol	Meaning
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; 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.
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 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.
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.
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 MIPS32™ 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

The MIPS Architecture: An Introduction

2.1 MIPS32 and MIPS64 Overview

2.1.1 Historical Perspective

The MIPS® Instruction Set Architecture (ISA) has evolved over time from the original MIPS I™ ISA, through the MIPS V™ ISA, to the current MIPS32™ and MIPS64™ Architectures. As the ISA evolved, all extensions have been backward compatible with previous versions of the ISA. In the MIPS III™ level of the ISA, 64-bit integers and addresses were added to the instruction set. The MIPS IV™ and MIPS V™ levels of the ISA added improved floating point operations, as well as a set of instructions intended to improve the efficiency of generated code and of data movement. Because of the strict backward-compatible requirement of the ISA, such changes were unavailable to 32-bit implementations of the ISA which were, by definition, MIPS I™ or MIPS II™ implementations.

While the user-mode ISA was always backward compatible, the privileged environment was allowed to change on a per-implementation basis. As a result, the R3000® privileged environment was different from the R4000® privileged environment, and subsequent implementations, while similar to the R4000 privileged environment, included subtle differences. Because the privileged environment was never part of the MIPS ISA, an implementation had the flexibility to make changes to suit that particular implementation. Unfortunately, this required kernel software changes to every operating system or kernel environment on which that implementation was intended to run.

Many of the original MIPS implementations were targeted at computer-like applications such as workstations and servers. In recent years MIPS implementations have had significant success in embedded applications. Today, most of the MIPS parts that are shipped go into some sort of embedded application. Such applications tend to have different trade-offs than computer-like applications including a focus on cost of implementation, and performance as a function of cost and power.

The MIPS32 and MIPS64 Architectures are intended to address the need for a high-performance but cost-sensitive MIPS instruction set. The MIPS32 Architecture is based on the MIPS II ISA, adding selected instructions from MIPS III, MIPS IV, and MIPS V to improve the efficiency of generated code and of data movement. The MIPS64 Architecture is based on the MIPS V ISA and is backward compatible with the MIPS32 Architecture. Both the MIPS32 and MIPS64 Architectures bring the privileged environment into the Architecture definition to address the needs of operating systems and other kernel software. Both also include provision for adding MIPS Application Specific Extensions (ASEs), User Defined Instructions (UDIs), and custom coprocessors to address the specific needs of particular markets.

MIPS32 and MIPS64 Architectures provides a substantial cost/performance advantage over microprocessor implementations based on traditional architectures. This advantage is a result of improvements made in several contiguous disciplines: VLSI process technology, CPU organization, system-level architecture, and operating system and compiler design.

2.2 Architectural Changes Relative to the MIPS I through MIPS V Architectures

In addition to the MIPS32 Architecture described in this document set, the following changes were made to the architecture relative to the earlier MIPS RISC Architecture Specification, which describes the MIPS I through MIPS V Architectures.

- The MIPS IV ISA added a restriction to the load and store instructions which have natural alignment requirements (all but load and store byte and load and store left and right) in which the base register used by the instruction must also be naturally aligned (the restriction expressed in the MIPS RISC Architecture Specification is that the offset be aligned, but the implication is that the base register is also aligned, and this is more consistent with the indexed load/store instructions which have no offset field). The restriction that the base register be naturally-aligned is eliminated by the MIPS32 Architecture, leaving the restriction that the effective address be naturally-aligned.

- Early MIPS implementations required two instructions separating a mflo or mfhi from the next integer multiply or divide operation. This hazard was eliminated in the MIPS IV ISA, although the MIPS RISC Architecture Specification does not clearly explain this fact. The MIPS32 Architecture explicitly eliminates this hazard and requires that the hi and lo registers be fully interlocked in hardware for all integer multiply and divide instructions (including, but not limited to, the madd, maddu, msub, msubu, and mul instructions introduced in this specification).
- The Implementation and Programming Notes included in the instruction descriptions for the madd, maddu, msub, msubu, and mul instructions should also be applied to all integer multiply and divide instructions in the MIPS RISC Architecture Specification.

2.2.1 MIPS Instruction Set Architecture (ISA)

The MIPS32 and MIPS64 Instruction Set Architectures define a compatible family of 32-bit and 64-bit instructions within the framework of the overall MIPS32 and MIPS64 Architectures. Included in the ISA are all instructions, both privileged and unprivileged, by which the programmer interfaces with the processor. The ISA guarantees object code compatibility for unprivileged and, often, privileged programs executing on any MIPS32 or MIPS64 processor; all instructions in the MIPS64 ISA are backward compatible with those instructions in the MIPS32 ISA. Using conditional compilation or assembly language macros, it is often possible to write privileged programs that run on both MIPS32 and MIPS64 implementations.

2.2.2 MIPS Privileged Resource Architecture (PRA)

The MIPS32 and MIPS64 Privileged Resource Architecture defines a set of environments and capabilities on which the ISA operates. The effects of some components of the PRA are visible to unprivileged programs; for instance, the virtual memory layout. Many other components are visible only to privileged programs and the operating system. The PRA provides the mechanisms necessary to manage the resources of the processor: virtual memory, caches, exceptions, user contexts, etc.

2.2.3 MIPS Application Specific Extensions (ASEs)

The MIPS32 and MIPS64 Architectures provide support for optional application specific extensions. As optional extensions to the base architecture, the ASEs do not burden every implementation of the architecture with instructions or capability that are not needed in a particular market. An ASE can be used with the appropriate ISA and PRA to meet the needs of a specific application or an entire class of applications.

2.2.4 MIPS User Defined Instructions (UDIs)

In addition to support for ASEs as described above, the MIPS32 and MIPS64 Architectures define specific instructions for the use of each implementation. The *Special2* instruction function fields and Coprocessor 2 are reserved for capability defined by each implementation.

2.3 Architecture Versus Implementation

When describing the characteristics of MIPS processors, *architecture* must be distinguished from the hardware *implementation of that architecture*.

- **Architecture** refers to the instruction set, registers and other state, the exception model, memory management, virtual and physical address layout, and other features that all hardware executes.
- **Implementation** refers to the way in which specific processors apply the architecture.

Here are two examples:

1. A floating point unit (FPU) is an optional part of the MIPS32 Architecture. A compatible implementation of the FPU may have different pipeline lengths, different hardware algorithms for performing multiplication or division, etc.
2. Most MIPS processors have caches; however, these caches are not implemented in the same manner in all MIPS processors. Some processors implement physically-indexed, physically tagged caches. Other implement virtually-indexed, physically-tagged caches. Still other processor implement more than one level of cache.

The MIPS32 architecture is decoupled from specific hardware implementations, leaving microprocessor designers free to create their own hardware designs within the framework of the architectural definition.

2.4 Relationship between the MIPS32 and MIPS64 Architectures

The MIPS Architecture evolved as a compromise between software and hardware resources. The architecture guarantees object-code compatibility for User-Mode programs executed on any MIPS processor. In User Mode MIPS64 processors are backward-compatible with their MIPS32 predecessors. As such, the MIPS32 Architecture is a strict subset of the MIPS64 Architecture. The relationship between the architectures is shown in Figure 2-1.

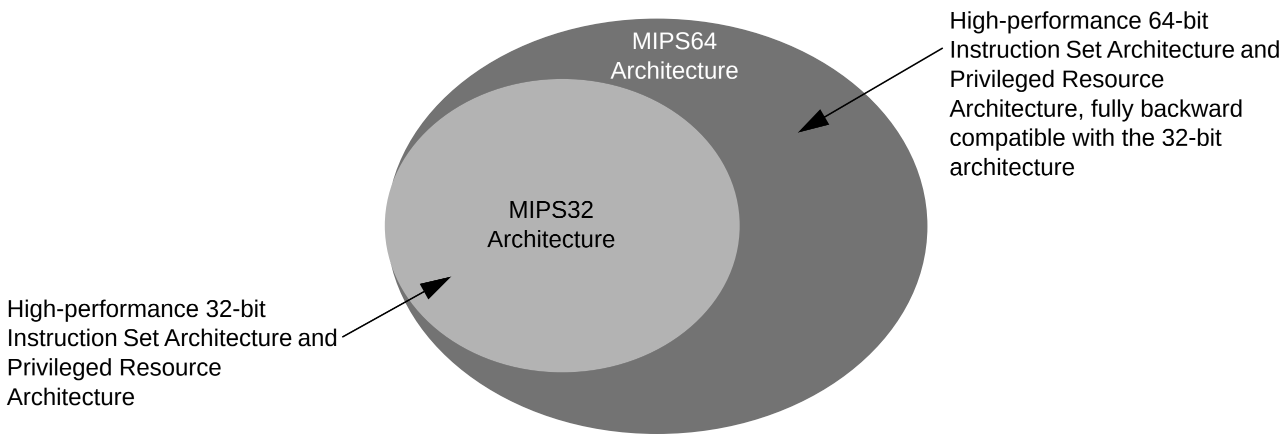

MIPS64 Architecture
High-performance 64-bit Instruction Set Architecture and Privileged Resource Architecture, fully backward compatible with the 32-bit architecture
MIPS32 Architecture
High-performance 32-bit Instruction Set Architecture and Privileged Resource Architecture

Figure 2-1 Relationship between the MIPS32 and MIPS64 Architectures

2.5 Instructions, Sorted by ISA

This section lists the instructions that are a part of the MIPS32 and MIPS64 ISAs.

2.5.1 List of MIPS32 Instructions

Table 2-1 lists of those instructions included in the MIPS32 ISA.

Table 2-1 MIPS32 Instructions

ABS.D	ABS.S	ADD	ADD.D	ADD.S	ADDI	ADDIU	ADDU
AND	ANDI	BC1F	BC1FL	BC1T	BC1TL	BC2F	BC2FL
BC2T	BC2TL	BEQ	BEQL	BGEZ	BGEZAL	BGEZALL	BGEZL
BGTZ	BGTZL	BLEZ	BLEZL	BLTZ	BLTZAL	BLTZALL	BLTZL
BNE	BNEL	BREAK	C.cond.D	C.cond.S	CACHE	CEIL.W.D	CEIL.W.S
CFC1	CFC2	CLO	CLZ	COP2	CTC1	CTC2	CVT.D.S
CVT.D.W	CVT.S.D	CVT.S.W	CVT.W.D	CVT.W.S	DIV	DIV.D	DIV.S
DIVU	ERET	FLOOR.W.D	FLOOR.W.S	J	JAL	JALR	JR
LB	LBU	LDC1	LDC2	LH	LHU	LL	LUI
LW	LWC1	LWC2	LWL	LWR	MADD	MADDU	MFC0
MFC1	MFC2	MFHI	MFLO	MOV.D	MOV.S	MOVF	MOVF.D
MOVF.S	MOVN	MOVN.D	MOVN.S	MOVT	MOVT.D	MOVT.S	MOVZ
MOVZ.D	MOVZ.S	MSUB	MSUBU	MTC0	MTC1	MTC2	MTHI
MTLO	MUL	MUL.D	MUL.S	MULT	MULTU	NEG.D	NEG.S
NOR	OR	ORI	PREF	ROUND.W.D	ROUND.W.S	SB	SC
SDC1	SDC2	SH	SLL	SLLV	SLT	SLTI	SLTIU
SLTU	SQRT.D	SQRT.S	SRA	SRAV	SRL	SRLV	SSNOP
SUB	SUB.D	SUB.S	SUBU	SW	SWC1	SWC2	SWL
SWR	SYNC	SYSCALL	TEQ	TEQI	TGE	TGEI	TGEIU
TGEU	TLBP	TLBR	TLBWI	TLBWR	TLT	TLTI	TLTIU
TLTU	TNE	TNEI	TRUNC.W.D	TRUNC.W.S	WAIT	XOR	XORI

2.5.2 List of MIPS64 Instructions

Table 2-2 lists of those instructions introduced in the MIPS64 ISA.

Table 2-2 MIPS64 Instructions

ABS.PS	ADD.PS	ALNV.PS	C.cond.PS	CEIL.L.D	CEIL.L.S	CVT.D.L	CVT.L.D
CVT.L.S	CVT.PS.S	CVT.S.L	CVT.S.PL	CVT.S.PU	DADD	DADDI	DADDIU
DADDU	DCLO	DDIV	DDIVU	DLCZ	DMFC0	DMFC1	DMFC2
DMTC0	DMTC1	DMTC2	DMULT	DMULTU	DSLL	DSLL32	DSLLV
DSRA	DSRA32	DSRAV	DSRL	DSRL32	DSRLV	DSUB	DSUBU

Table 2-2 MIPS64 Instructions

FLOOR.L.D	FLOOR.L.S	LD	LDL	LDR	LDXC1	LLD	LUXC1
LWU	LWXC1	MADD.D	MADD.PS	MADD.S	MOV.PS	MOVF.PS	MOVN.PS
MOVT.PS	MOVZ.PS	MSUB.D	MSUB.PS	MSUB.S	MUL.PS	NEG.PS	NMADD.D
NMADD.PS	NMADD.S	NMSUB.D	NMSUB.PS	NMSUB.S	PLL.PS	PLU.PS	PREFX
PUL.PS	PUU.PS	RECIP.D	RECIP.S	ROUND.L.D	ROUND.L.S	RSQRT.D	RSQRT.S
SCD	SD	SDL	SDR	SDXC1	SUB.PS	SUXC1	SWXC1
TRUNC.L.D	TRUNC.L.S						

2.6 Pipeline Architecture

This section describes the basic pipeline architecture, along with two types of improvements: superpipelines and superscalar pipelines. (Pipelining and multiple issuing are not defined by the ISA, but are implementation dependent.)

2.6.1 Pipeline Stages and Execution Rates

MIPS processors all use some variation of a pipeline in their architecture. A pipeline is divided into the following discrete parts, or **stages**, shown in Figure 2-2:

- Fetch
- Arithmetic operation
- Memory access
- Write back

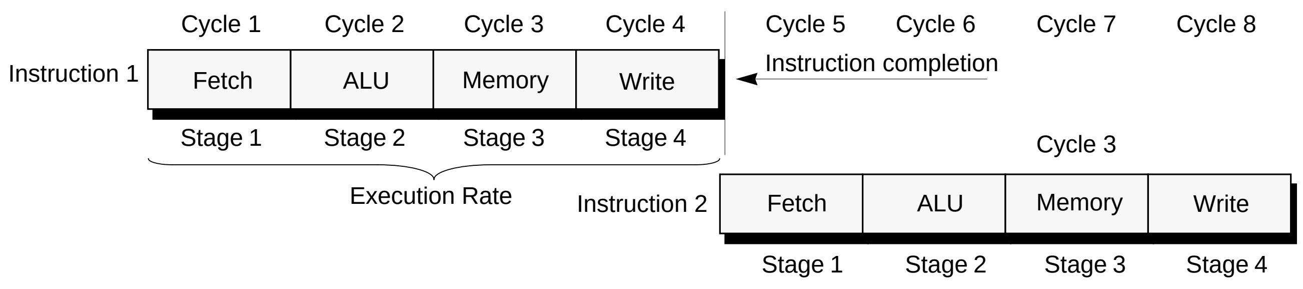


Figure 2-2 One-Deep Single-Completion Instruction Pipeline

In the example shown in Figure 2-2, each stage takes one processor clock cycle to complete. Thus it takes four clock cycles (ignoring delays or stalls) for the instruction to complete. In this example, the **execution rate** of the pipeline is one instruction every four clock cycles. Conversely, because only a single execution can be fetched before completion, only one stage is active at any time.

2.6.2 Parallel Pipeline

Figure 2-3 illustrates a remedy for the **latency** (the time it takes to execute an instruction) inherent in the pipeline shown in Figure 2-2.

Instead of waiting for an instruction to be completed before the next instruction can be fetched (four clock cycles), a new instruction is fetched each clock cycle. There are four stages to the pipeline so the four instructions can be executed simultaneously, one at each stage of the pipeline. It still takes four clock cycles for the first instruction to be completed; however, in this theoretical example, a new instruction is completed every clock cycle thereafter. Instructions in Figure 2-3 are executed at a rate four times that of the pipeline shown in Figure 2-2.

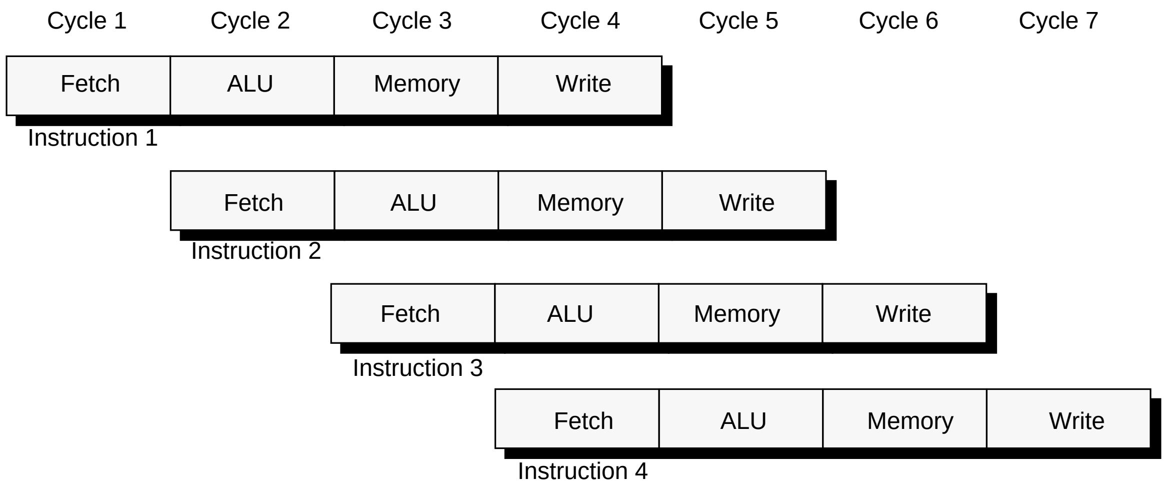


Figure 2-3 Four-Deep Single-Completion Pipeline

2.6.3 Superpipeline

Figure 2-4 shows a **superpipelined** architecture. Each stage is designed to take only a fraction of an external clock cycle—in this case, half a clock. Effectively, each stage is divided into more than one **substage**. Therefore more than one instruction can be completed each cycle.

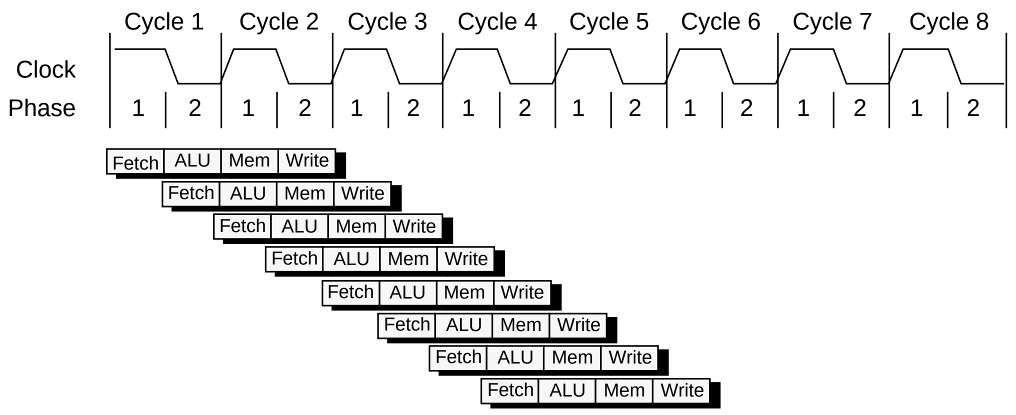
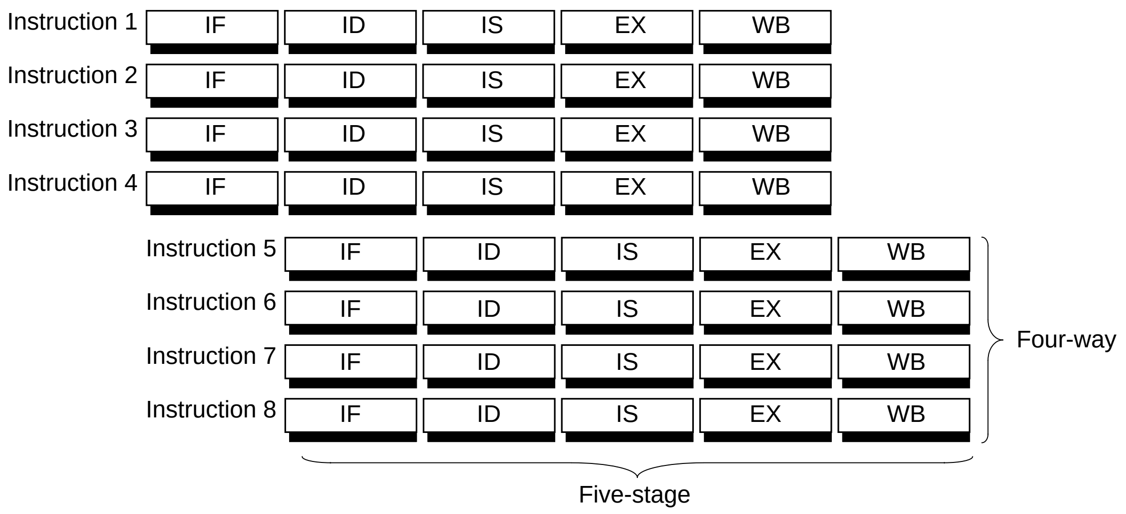


Figure 2-4 Four-Deep Superpipeline

2.6.4 Superscalar Pipeline

A **superscalar** architecture also allows more than one instruction to be completed each clock cycle. Figure 2-5 shows a four-way, five-stage superscalar pipeline.



IF = instruction fetch
ID = instruction decode and dependency
IS = instruction issue
EX = execution
WB = write back

Figure 2-5 Four-Way Superscalar Pipeline

2.7 Load/Store Architecture

Generally, it takes longer to perform operations in memory than it does to perform them in on-chip registers. This is because of the difference in time it takes to access a register (fast) and main memory (slower).

To eliminate the longer access time, or **latency**, of in-memory operations, MIPS processors use a **load/store** design. The processor has many registers on chip, and all operations are performed on operands held in these processor registers. Main memory is accessed only through load and store instructions. This has several benefits:

- Reducing the number of memory accesses, easing memory bandwidth requirements
- Simplifying the instruction set
- Making it easier for compilers to optimize register allocation

2.8 Programming Model

This section describes the following aspects of the programming model:

- “CPU Data Formats”
- “Coprocessors (CP0-CP3)”
- “CPU Registers”
- “FPU Data Formats”
- “Byte Ordering and Endianness”
- “Memory Access Types”

2.8.1 CPU Data Formats

The CPU defines the following data formats:

- Bit (*b*)
- Byte (8 bits, *B*)
- Halfword (16 bits, *H*)
- Word (32 bits, *W*)
- Doubleword (64 bits, *D*)[1]

2.8.2 FPU Data Formats

The FPU defines the following data formats:

- 32-bit single-precision floating point (.fmt type *S*)
- 32-bit single-precision floating point paired-single (.fmt type *PS*)[1]
- 64-bit double-precision floating point (.fmt type *D*)
- 32-bit Word fixed point (.fmt type *W*)
- 64-bit Long fixed point (.fmt type *L*)[1]

2.8.3 Coprocessors (CP0-CP3)

The MIPS Architecture defines four coprocessors (designated CP0, CP1, CP2, and CP3):

- Coprocessor 0 (**CP0**) is incorporated on the CPU chip and supports the virtual memory system and exception handling. CP0 is also referred to as the *System Control Coprocessor.*
- Coprocessor 1 (**CP1**) is reserved for the floating point coprocessor, the FPU.
- Coprocessor 2 (**CP2**) is available for specific implementations.
- Coprocessor 3 (**CP3**) is reserved for the floating point unit in the MIPS64 Architecture.

CP0 translates virtual addresses into physical addresses, manages exceptions, and handles switches between kernel, supervisor, and user states. CP0 also controls the cache subsystem, as well as providing diagnostic control and error recovery facilities. The architectural features of CP0 are defined in Volume III.

2.8.4 CPU Registers

The MIPS32 Architecture defines the following CPU registers:

- 32 32-bit general purpose registers (GPRs)
- a pair of special-purpose registers to hold the results of integer multiply, divide, and multiply-accumulate operations (HI and LO)
- a special-purpose program counter (PC), which is affected only indirectly by certain instructions - it is not an architecturally-visible register.

[1] The CPU Doubleword and FPU floating point paired-single and and Long fixed point data formats are available only in the MIPS64 Architecture

2.8.4.1 CPU General-Purpose Registers

Two of the CPU general-purpose registers have assigned functions:

- *r0* is hard-wired to a value of zero, and can be used as the target register for any instruction whose result is to be discarded. *r0* can also be used as a source when a zero value is needed.
- *r31* is the destination register used by JAL, BLTZAL, BLTZALL, BGEZAL, and BGEZALL without being explicitly specified in the instruction word. Otherwise *r31* is used as a normal register.

The remaining registers are available for general-purpose use.

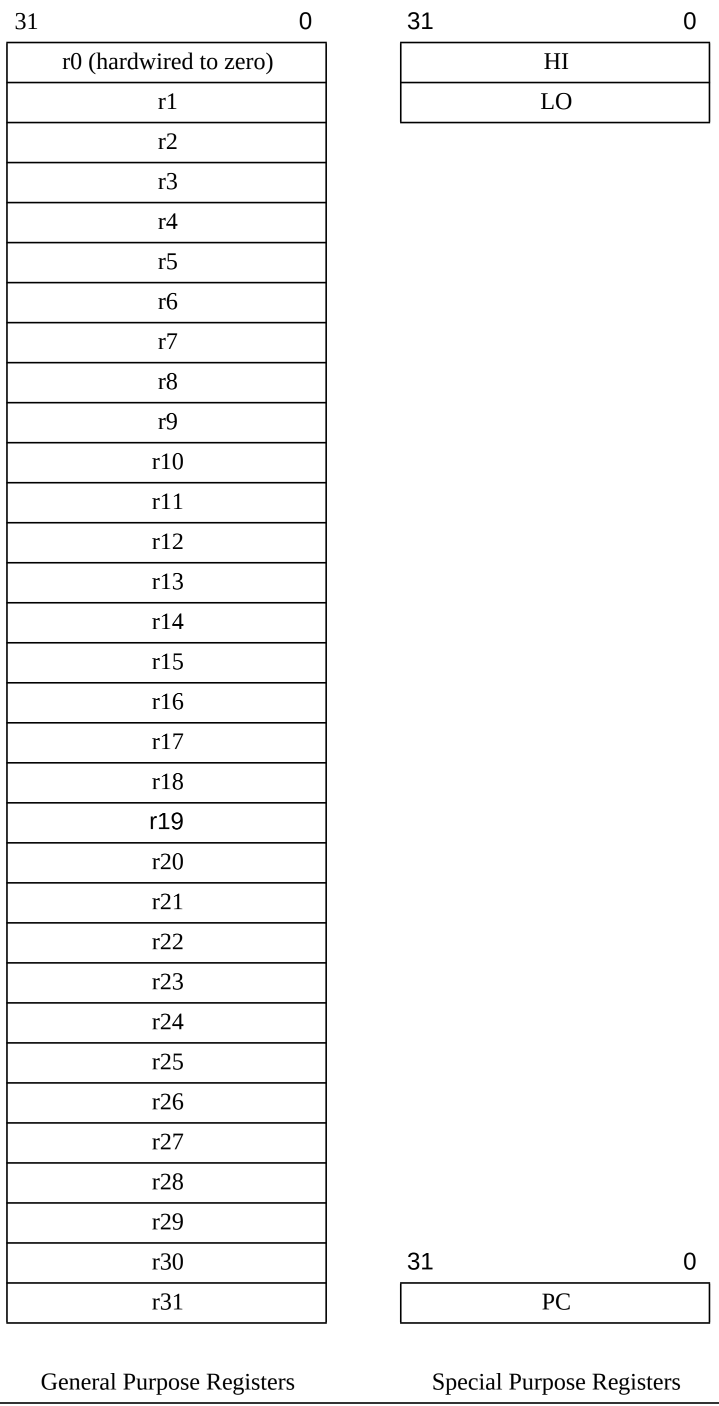
2.8.4.2 CPU Special-Purpose Registers

The CPU contains three special-purpose registers:

- *PC*—Program Counter register
- *HI*—Multiply and Divide register higher result
- *LO*—Multiply and Divide register lower result
  - During a multiply operation, the *HI* and *LO* registers store the product of integer multiply.
  - During a multiply-add or multiply-subtract operation, the *HI* and *LO* registers store the result of the integer multiply-add or multiply-subtract.
  - During a division, the *HI* and *LO* registers store the quotient (in *LO*) and remainder (in *HI*) of integer divide.
  - During a multiply-accumulate, the *HI* and *LO* registers store the accumulated result of the operation.

Figure 2-6 shows the layout of the CPU registers.

Figure 2-6 CPU Registers



2.8.5 FPU Registers

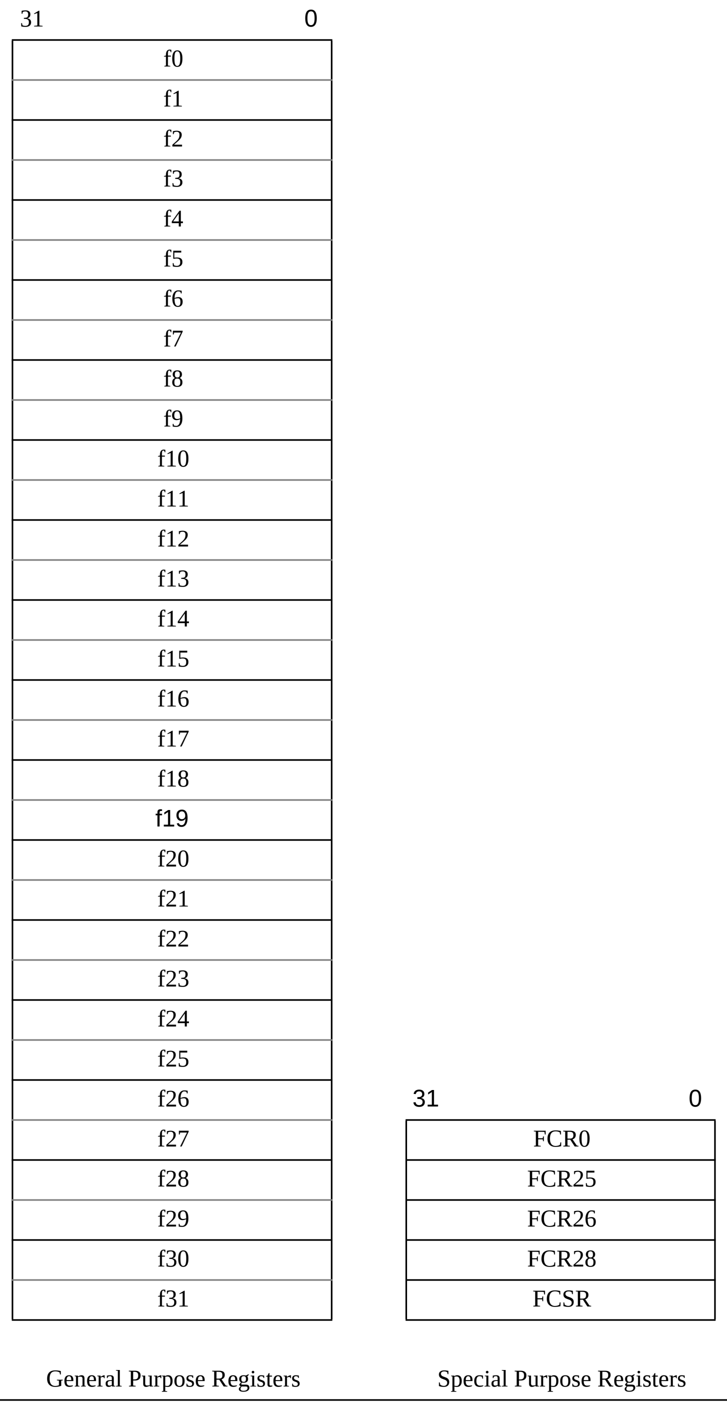
The MIPS32 Architecture defines the following FPU registers:

- 32 32-bit floating point registers (FPRs). All 32 registers are available for use in single-precision floating point operations. Double-precision floating point values are stored in even-odd pairs of FPRs.
- Five FPU control registers are used to identify and control the FPU.

The MIPS32 ISA includes 8 floating point condition codes as part of the FCSR register in the floating point unit.

Figure 2-7 shows the layout of the FPU Registers.

Figure 2-7 FPU Registers



2.8.6 Byte Ordering and Endianness

Bytes within larger CPU data formats—halfword, word, and doubleword—can be configured in either big-endian or little-endian order, as described in the following subsections:

- “Big-Endian Order”
- “Little-Endian Order”
- “MIPS Bit Endianness”

Endianness defines the location of byte 0 within a larger data structure (in this book, bits are always numbered with 0 on the right). Figures 2-8 and 2-9 show the ordering of bytes within words and the ordering of words within multiple-word structures for both big-endian and little-endian configurations.

2.8.6.1 Big-Endian Order

When configured in **big-endian order**, byte 0 is the most-significant (left-hand) byte. Figure 2-8 shows this configuration.

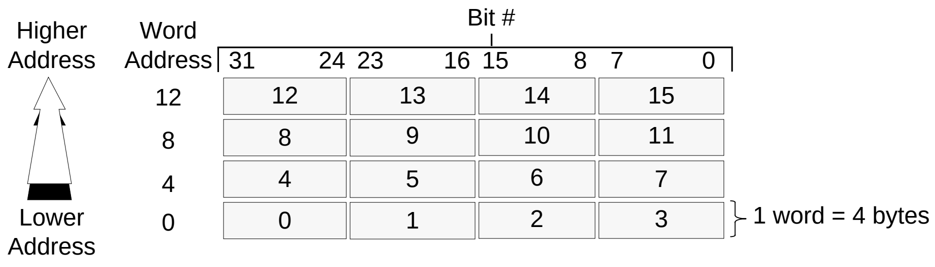


Figure 2-8 Big-Endian Byte Ordering

2.8.6.2 Little-Endian Order

When configured in **little-endian order**, byte 0 is always the least-significant (right-hand) byte. Figure 2-9 shows this configuration.

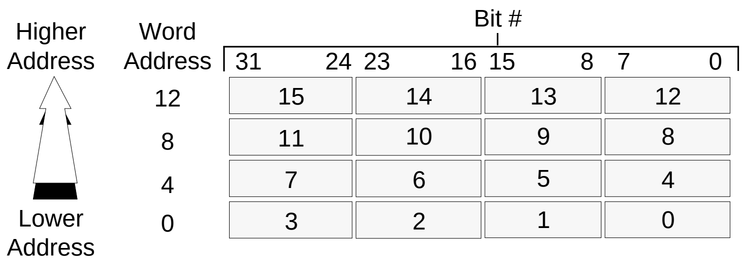


Figure 2-9 Little-Endian Byte Ordering

2.8.6.3 MIPS Bit Endianness

In this book, bit 0 is always the least-significant (right-hand) bit. Although no instructions explicitly designate bit positions within words, MIPS bit designations are always little-endian.

Figure 2-10 shows big-endian and Figure 2-11 shows little-endian byte ordering in doublewords.

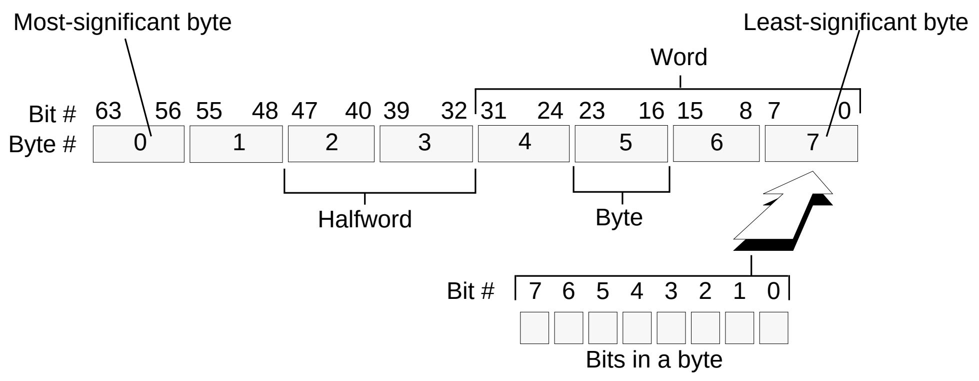


Figure 2-10 Big-Endian Data in Doubleword Format

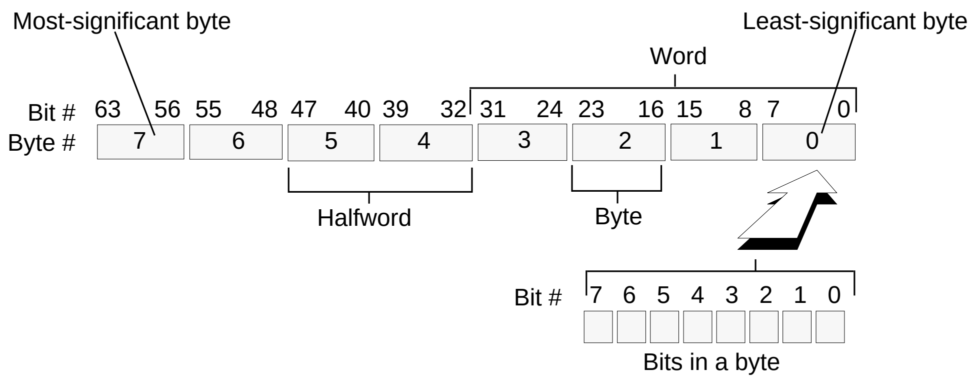


Figure 2-11 Little-Endian Data in Doubleword Format

2.8.6.4 Addressing Alignment Constraints

The CPU uses byte addressing for halfword, word, and doubleword accesses with the following alignment constraints:

- Halfword accesses must be aligned on an even byte boundary (0, 2, 4...).
- Word accesses must be aligned on a byte boundary divisible by four (0, 4, 8...).
- Doubleword accesses must be aligned on a byte boundary divisible by eight (0, 8, 16...).

2.8.6.5 Unaligned Loads and Stores

The following instructions load and store words that are not aligned on word (W) or doubleword (D) boundaries:

Table 2-3 Unaligned Load and Store Instructions

Alignment	Instructions	Instruction Set
Word	LWL, LWR, SWL, SWR	MIPS32 ISA
Doubleword	LDL, LDR, SDL, SDR	MIPS64 ISA

Figure 2-12 show a big-endian access of a misaligned word that has byte address 3, and Figure 2-13 shows a little-endian access of a misaligned word that has byte address 1.[1]

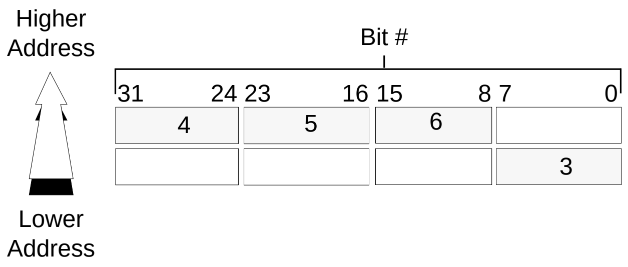


Figure 2-12 Big-Endian Misaligned Word Addressing

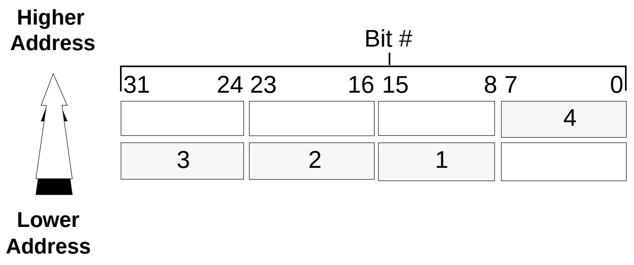


Figure 2-13 Little-Endian Misaligned Word Addressing

2.8.7 Memory Access Types

MIPS systems provide several *memory access types*. These are characteristic ways to use physical memory and caches to perform a memory access.

The **memory access type** is identified by the cache coherence algorithm (*CCA*) bits in the TLB entry for each mapped virtual page. The access type used for a location is associated with the virtual address, not the physical address or the instruction making the reference. Memory access types are available for both uniprocessor and multiprocessor (MP) implementations.

All implementations must provide the following memory access types:

- Uncached
- Cached

These memory access types are described in the following sections:

- "Uncached Memory Access"
- "Cached Memory Access"

2.8.7.1 Uncached Memory Access

In an *uncached* access, physical memory resolves the access. Each reference causes a read or write to physical memory. Caches are neither examined nor modified.

[1] These two figures show left-side misalignment.

2.8.7.2 Cached Memory Access

In a *cached* access, physical memory and all caches in the system containing a copy of the physical location are used to resolve the access. A copy of a location is coherent if the copy was placed in the cache by a *cached coherent* access; a copy of a location is noncoherent if the copy was placed in the cache by a *cached noncoherent* access. (Coherency is dictated by the system architecture, not the processor implementation.)

Caches containing a coherent copy of the location are examined and/or modified to keep the contents of the location coherent. It is not possible to predict whether caches holding a noncoherent copy of the location will be examined and/or modified during a *cached coherent* access.

2.8.8 Implementation-Specific Access Types

An implementation may provide memory access types other than *uncached* or *cached*. Implementation-specific documentation accompanies each processor, and defines the properties of the new access types and their effect on all memory-related operations.

2.8.9 Cache Coherence Algorithms and Access Types

Memory access types are specified by architecturally-defined and implementation-specific cache coherence algorithm bits (*CCA*s) kept in TLB entries.

Slightly different cache coherence algorithms such as “cached coherent, update on write” and “cached coherent, exclusive on write” can map to the same memory access type; in this case they both map to *cached coherent*. In order to map to the same access type, the fundamental mechanisms of both *CCA*s must be the same.

When the operation of the instruction is affected, the instructions are described in terms of memory access types. The load and store operations in a processor proceed according to the specific *CCA* of the reference, however, and the pseudocode for load and store common functions uses the *CCA* value rather than the corresponding memory access type.

2.8.10 Mixing Access Types

It is possible to have more than one virtual location mapped to the same physical location (known as **aliasing**). The memory access type used for the virtual mappings may be different, but it is not generally possible to use mappings with different access types at the same time.

For all accesses to virtual locations with the *same* memory access type, a processor executing load and store instructions on a physical location must ensure that the instructions occur in proper program order.

A processor can execute a load or store to a physical location using one access type, but any subsequent load or store to the same location using a different memory access type is **UNPREDICTABLE**, unless a privileged instruction sequence to change the access type is executed between the two accesses. Each implementation has a privileged implementation-specific mechanism to change access types.

The memory access type of a location affects the behavior of I-fetch, load, store, and prefetch operations to that location. In addition, memory access types affect some instruction descriptions. Load Linked (LL, LLD) and Store Conditional (SC, SCD) have defined operation only for locations with *cached* memory access type.

Chapter 3

Application Specific Extensions

This section gives an overview of the Architecture Specific Extensions that are supported by the MIPS32 Architecture.

3.1 Description of ASEs

As the MIPS architecture is adopted into a wider variety of markets, the need to extend this architecture in different directions becomes more and more apparent. Therefore various optional application-specific extensions are provided for use with the base ISAs (MIPS32 and MIPS64). The ASEs are optional, so the architecture is not permanently bound to support them and the ASEs are used only as needed.

Extensions to the ISA are driven by the requirements of the computer segment, or by customers whose focus is primarily on performance. An ASE can be used with the appropriate ISA to meet the needs of a specific application or an entire class of applications.

Figure 3-1 shows how ASEs interrelate with ISAs.

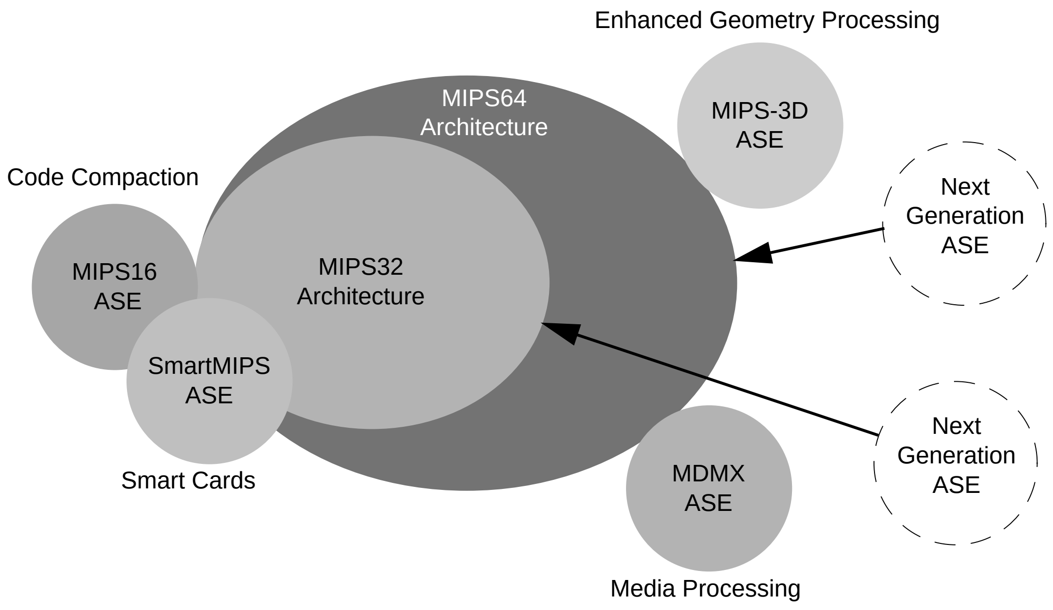


Figure 3-1 MIPS ISAs and ASEs

Figure 3-2 User-Mode MIPS ISAs and Optional ASEs

The MIPS32 Architecture is a strict subset of the MIPS64 Architecture. ASEs are applicable to one or both of the base architectures as dictated by market need and the requirements placed on the base architecture by the ASE definition.

3.2 List of Application Specific Instructions

As of the publishing date of this document, the following Application Specific Extensions were supported by the architecture.

ASE	Base Architecture Requirement	Use
MIPS16™	MIPS32 or MIPS64	Code Compaction
MDMX™	MIPS64	Digital Media
MIPS-3D™	MIPS64	Geometry Processing
SmartMIPS™	MIPS32	Smart Cards and Smart Objects

3.2.1 The MIPS16 Application Specific Extension to the MIPS32Architecture

The MIPS16 ASE is composed of 16-bit compressed code instructions, designed for the embedded processor market and situations with tight memory constraints. The core can execute both 16- and 32-bit instructions intermixed in the same program, and is compatible with both the MIPS32 and MIPS64 Architectures. Volume IV-a of this document set describes the MIPS16 ASE.

3.2.2 The MDMX Application Specific Extension to the MIPS64 Architecture

The MIPS Digital Media Extension (MDMX) provides video, audio, and graphics pixel processing through vectors of small integers. Although not a part of the MIPS ISA, this extension is included for informational purposes. Because the MDMX ASE requires the MIPS64 Architecture, it is not discussed in this document set.

3.2.3 The MIPS-3D Application Specific Extension to the MIPS64 Architecture

The MIPS-3D ASE provides enhanced performance of geometry processing calculations by building on the paired single floating point data type, and adding specific instructions to accelerate computations on these data types. Because the MIPS-3D ASE requires the MIPS64 Architecture, it is not discussed in this document set.

3.2.4 The SmartMIPS Application Specific Extension to the MIPS32 Architecture

The SmartMIPS ASE extends the MIPS32 Architecture with a set of new and modified instruction designed to improve the performance and reduce the memory consumption of MIPS-based smart card or smart object systems. Volume IV-d of this document set describes the SmartMIPS ASE.

Chapter 4

Overview of the CPU Instruction Set

This chapter gives an overview of the CPU instructions, including a description of CPU instruction formats. An overview of the FPU instructions is given in Chapter 5.

4.1 CPU Instructions, Grouped By Function

CPU instructions are organized into the following functional groups:

- Load and store
- Computational
- Jump and branch
- Miscellaneous
- Coprocessor

Each instruction is 32 bits long.

4.1.1 CPU Load and Store Instructions

MIPS processors use a load/store architecture; all operations are performed on operands held in processor registers and main memory is accessed only through load and store instructions.

4.1.1.1 Types of Loads and Stores

There are several different types of load and store instructions, each designed for a different purpose:

- Transferring variously-sized fields (for example, LB, SW)
- Trading transferred data as signed or unsigned integers (for example, LHU)
- Accessing unaligned fields (for example, LWR, SWL)
- Atomic memory update (read-modify-write: for instance, LL/SC)

Regardless of the byte ordering (big- or little-endian), the address of a halfword, or word is the lowest byte address among the bytes forming the object:

- For big-endian ordering, this is the most-significant byte.
- For a little-endian ordering, this is the least-significant byte.

Refer to “Byte Ordering and Endianness” on page 17 for more information on big-endian and little-endian data ordering.

4.1.1.2 Load and Store Access Types

Table 4-1 lists the data sizes that can be accessed through CPU load and store operations. These tables also indicate the particular ISA within which each operation is defined.

Table 4-1 Load and Store Operations Using Register + Offset Addressing Mode

Data Size	CPU			Coprocessors 1 and 2	
	Load Signed	Load Unsigned	Store	Load	Store
Byte	MIPS32	MIPS32	MIPS32		
Halfword	MIPS32	MIPS32	MIPS32		
Word	MIPS32	MIPS64	MIPS32	MIPS32	MIPS32
Unaligned word	MIPS32		MIPS32		
Linked word (atomic modify)	MIPS32		MIPS32		

4.1.1.3 List of CPU Load and Store Instructions

The following data sizes (as defined in the *AccessLength* field) are transferred by CPU load and store instructions:

- Byte
- Halfword
- Word

Signed and unsigned integers of different sizes are supported by loads that either sign-extend or zero-extend the data loaded into the register.

Table 4-2 lists aligned CPU load and store instructions, while unaligned loads and stores are listed in Table 4-3. Each table also lists the MIPS ISA within which an instruction is defined.

Table 4-2 Aligned CPU Load/Store Instructions

Mnemonic	Instruction	Defined in MIPS ISA
LB	Load Byte	MIPS32
LBU	Load Byte Unsigned	MIPS32
LH	Load Halfword	MIPS32
LHU	Load Halfword Unsigned	MIPS32
LW	Load Word	MIPS32
SB	Store Byte	MIPS32
SH	Store Halfword	MIPS32
SW	Store Word	MIPS32

Unaligned words and doublewords can be loaded or stored in just two instructions by using a pair of the special instructions listed in Table 4-3. The load instructions read the left-side or right-side bytes (left or right side of register) from an aligned word and merge them into the correct bytes of the destination register.

Unaligned CPU load and store instructions are listed in Table 4-3, along with the MIPS ISA within which an instruction is defined.

Table 4-3 Unaligned CPU Load and Store Instructions

Mnemonic	Instruction	Defined in MIPS ISA
LWL	Load Word Left	MIPS32
LWR	Load Word Right	MIPS32
SWL	Store Word Left	MIPS32
SWR	Store Word Right	MIPS32

4.1.1.4 Loads and Stores Used for Atomic Updates

The paired instructions, Load Linked and Store Conditional, can be used to perform an atomic read-modify-write of word or doubleword cached memory locations. These instructions are used in carefully coded sequences to provide one of several synchronization primitives, including test-and-set, bit-level locks, semaphores, and sequencers and event counts. Table 4-4 lists the LL and SC instructions, along with the MIPS ISA within which an instruction is defined.

Table 4-4 Atomic Update CPU Load and Store Instructions

Mnemonic	Instruction	Defined in MIPS ISA
LL	Load Linked Word	MIPS32
SC	Store Conditional Word	MIPS32

4.1.1.5 Coprocessor Loads and Stores

If a particular coprocessor is not enabled, loads and stores to that processor cannot execute and the attempted load or store causes a Coprocessor Unusable exception. Enabling a coprocessor is a privileged operation provided by the System Control Coprocessor, CP0.

Table 4-5 lists the coprocessor load and store instructions.

Table 4-5 Coprocessor Load and Store Instructions

Mnemonic	Instruction	Defined in MIPS ISA
LDCz	Load Doubleword to Coprocessor-z, z = 1 or 2	MIPS32
LWCz	Load Word to Coprocessor-z, z = 1 or 2	MIPS32
SDCz	Store Doubleword from Coprocessor-z, z = 1 or 2	MIPS32
SWCz	Store Word from Coprocessor-z, z = 1 or 2	MIPS32

4.1.2 Computational Instructions

This section describes the following:

- “ALU Immediate and Three-Operand Instructions”
- “ALU Two-Operand Instructions”
- “Shift Instructions”
- “Multiply and Divide Instructions”

2’s complement arithmetic is performed on integers represented in 2’s complement notation. These are signed versions of the following operations:

- Add
- Subtract
- Multiply
- Divide

The add and subtract operations labelled “unsigned” are actually modulo arithmetic without overflow detection.

There are also unsigned versions of *multiply* and *divide*, as well as a full complement of *shift* and *logical* operations. Logical operations are not sensitive to the width of the register.

MIPS32 provided 32-bit integers and 32-bit arithmetic.

4.1.2.1 ALU Immediate and Three-Operand Instructions

Table 4-6 lists those arithmetic and logical instructions that operate on one operand from a register and the other from a 16-bit *immediate* value supplied by the instruction word. This table also lists the MIPS ISA within which an instruction is defined.

The *immediate* operand is treated as a signed value for the arithmetic and compare instructions, and treated as a logical value (zero-extended to register length) for the logical instructions.

Table 4-6 ALU Instructions With an Immediate Operand

Mnemonic	Instruction	Defined in MIPS ISA
ADDI	Add Immediate Word	MIPS32
ADDIU[a]	Add Immediate Unsigned Word	MIPS32
ANDI	And Immediate	MIPS32
LUI	Load Upper Immediate	MIPS32
ORI	Or Immediate	MIPS32
SLTI	Set on Less Than Immediate	MIPS32
SLTIU	Set on Less Than Immediate Unsigned	MIPS32
XORI	Exclusive Or Immediate	MIPS32

a. The term “unsigned” in the instruction name is a misnomer; this operation is 32-bit modulo arithmetic that does not trap on overflow.

Table 4-7 describes ALU instructions that use three operands, along with the MIPS ISA within which an instruction is defined.

Table 4-7 Three-Operand ALU Instructions

Mnemonic	Instruction	Defined in MIPS ISA
ADD	Add Word	MIPS32
ADDU[a]	Add Unsigned Word	MIPS32
AND	And	MIPS32
NOR	Nor	MIPS32

Table 4-7 Three-Operand ALU Instructions (Continued)

Mnemonic	Instruction	Defined in MIPS ISA
OR	Or	MIPS32
SLT	Set on Less Than	MIPS32
SLTU	Set on Less Than Unsigned	MIPS32
SUB	Subtract Word	MIPS32
SUBU[a]	Subtract Unsigned Word	MIPS32
XOR	Exclusive Or	MIPS32

a. The term “unsigned” in the instruction name is a misnomer; this operation is 32-bit modulo arithmetic that does not trap on overflow.

4.1.2.2 ALU Two-Operand Instructions

Table 4-7 describes ALU instructions that use two operands, along with the MIPS ISA within which an instruction is defined.

Table 4-8 Three-Operand ALU Instructions

Mnemonic	Instruction	Defined in MIPS ISA
CLO	Count Leading Ones in Word	MIPS32
CLZ	Count Leading Zeros in Word	MIPS32
NOR	Nor	MIPS32
OR	Or	MIPS32
XOR	Exclusive Or	MIPS32

4.1.2.3 Shift Instructions

The ISA defines two types of shift instructions:

- Those that take a fixed shift amount from a 5-bit field in the instruction word (for instance, SLL, SRL)
- Those that take a shift amount from the low-order bits of a general register (for instance, SRAV, SRLV)

Shift instructions are listed in Table 4-9, along with the MIPS ISA within which an instruction is defined.

Table 4-9 Shift Instructions

Mnemonic	Instruction	Defined in MIPS ISA
SLL	Shift Word Left Logical	MIPS32
SLLV	Shift Word Left Logical Variable	MIPS32
SRA	Shift Word Right Arithmetic	MIPS32
SRAV	Shift Word Right Arithmetic Variable	MIPS32
SRL	Shift Word Right Logical	MIPS32
SRLV	Shift Word Right Logical Variable	MIPS32

4.1.2.4 Multiply and Divide Instructions

The multiply and divide instructions produce twice as many result bits as is typical with other processors. With one exception, they deliver their results into the *HI* and *LO* special registers. The MUL instruction delivers the lower half of the result directly to a GPR.

- **Multiply** produces a full-width product twice the width of the input operands; the low half is loaded into *LO* and the high half is loaded into *HI*.
- **Multiply-Add** and **Multiply-Subtract** produce a full-width product twice the width of the input operations and adds or subtracts the product from the concatenated value of *HI* and *LO*. The low half of the addition is loaded into *LO* and the high half is loaded into *HI*.
- **Divide** produces a quotient that is loaded into *LO* and a remainder that is loaded into *HI*.

The results are accessed by instructions that transfer data between *HI*/*LO* and the general registers.

Table 4-10 lists the multiply, divide, and *HI*/*LO* move instructions, along with the MIPS ISA within which an instruction is defined.

Table 4-10 Multiply/Divide Instructions

Mnemonic	Instruction	Defined in MIPS ISA
DIV	Divide Word	MIPS32
DIVU	Divide Unsigned Word	MIPS32
MADD	Multiply and Add Word	MIPS32
MADDU	Multiply and Add Word Unsigned	MIPS32
MFHI	Move From HI	MIPS32
MFLO	Move From LO	MIPS32
MSUB	Multiply and Subtract Word	MIPS32
MSUBU	Multiply and Subtract Word Unsigned	MIPS32
MTHI	Move To HI	MIPS32
MTLO	Move To LO	MIPS32
MUL	Multiply Word to Register	MIPS32
MULT	Multiply Word	MIPS32
MULTU	Multiply Unsigned Word	MIPS32

4.1.3 Jump and Branch Instructions

This section describes the following:

- “Types of Jump and Branch Instructions Defined by the ISA”
- “Branch Delays and the Branch Delay Slot”
- “Branch and Branch Likely”
- “List of Jump and Branch Instructions”

4.1.3.1 Types of Jump and Branch Instructions Defined by the ISA

The architecture defines the following jump and branch instructions:

- PC-relative conditional branch
- PC-region unconditional jump
- Absolute (register) unconditional jump
- A set of procedure calls that record a return link address in a general register.

4.1.3.2 Branch Delays and the Branch Delay Slot

All branches have an architectural delay of one instruction. The instruction immediately following a branch is said to be in the **branch delay slot**. If a branch or jump instruction is placed in the branch delay slot, the operation of both instructions is undefined.

By convention, if an exception or interrupt prevents the completion of an instruction in the branch delay slot, the instruction stream is continued by re-executing the branch instruction. To permit this, branches must be restartable; procedure calls may not use the register in which the return link is stored (usually GPR *31*) to determine the branch target address.

4.1.3.3 Branch and Branch Likely

There are two versions of conditional branches; they differ in the manner in which they handle the instruction in the delay slot when the branch is not taken and execution falls through.

- **Branch** instructions execute the instruction in the delay slot.
- **Branch likely** instructions do not execute the instruction in the delay slot if the branch is not taken (they are said to *nullify* the instruction in the delay slot).

  Although the Branch Likely instructions are included in this specification, software is strongly encouraged to avoid the use of the Branch Likely instructions, as they will be removed from a future revision of the MIPS Architecture.

4.1.3.4 List of Jump and Branch Instructions

Table 4-11 lists instructions that jump to a procedure call within the current 256 MB-aligned region, or to an absolute address held in a register.

Table 4-11 lists the unconditional jump instructions within a given 256 MByte region. Table 4-12 lists branch instructions that compare two registers before conditionally executing a PC-relative branch. Table 4-13 lists branch instructions that test a register—compare with zero—before conditionally executing a PC-relative branch. Table 4-14 lists the deprecated Branch Likely Instructions.

Each table also lists the MIPS ISA within which an instruction is defined.

Table 4-11 Unconditional Jump Within a 256 Megabyte Region

Mnemonic	Instruction	Location to Which Jump Is Made	Defined in MIPS ISA
J	Jump	256 Megabyte Region	MIPS32
JAL	Jump and Link	256 Megabyte Region	MIPS32
JALR	Jump and Link Register	Absolute Address	MIPS32

Table 4-11 Unconditional Jump Within a 256 Megabyte Region

Mnemonic	Instruction	Location to Which Jump Is Made	Defined in MIPS ISA
JALX	Jump and Link Exchange	Absolute Address	MIPS16
JR	Jump Register	Absolute Address	MIPS32

Table 4-12 PC-Relative Conditional Branch Instructions Comparing Two Registers

Mnemonic	Instruction	Defined in MIPS ISA
BEQ	Branch on Equal	MIPS32
BNE	Branch on Not Equal	MIPS32

Table 4-13 PC-Relative Conditional Branch Instructions Comparing With Zero

Mnemonic	Instruction	Defined in MIPS ISA
BGEZ	Branch on Greater Than or Equal to Zero	MIPS32
BGEZAL	Branch on Greater Than or Equal to Zero and Link	MIPS32
BGTZ	Branch on Greater Than Zero	MIPS32
BLEZ	Branch on Less Than or Equal to Zero	MIPS32
BLTZ	Branch on Less Than Zero	MIPS32
BLTZAL	Branch on Less Than Zero and Link	MIPS32

Table 4-14 Deprecated Branch Likely Instructions

Mnemonic	Instruction	Defined in MIPS ISA
BEQL	Branch on Equal Likely	MIPS32
BGEZALL	Branch on Greater Than or Equal to Zero and Link Likely	MIPS32
BGEZL	Branch on Greater Than or Equal to Zero Likely	MIPS32
BGTZL	Branch on Greater Than Zero Likely	MIPS32
BLEZL	Branch on Less Than or Equal to Zero Likely	MIPS32
BLTZALL	Branch on Less Than Zero and Link Likely	MIPS32
BLTZL	Branch on Less Than Zero Likely	MIPS32
BNEL	Branch on Not Equal Likely	MIPS32

4.1.4 Miscellaneous Instructions

Miscellaneous instructions include:

- “Instruction Serialization (SYNC)”
- “Exception Instructions”

- "Conditional Move Instructions"
- "Prefetch Instructions"
- "NOP Instructions"

4.1.4.1 Instruction Serialization (SYNC)

In normal operation, the order in which load and store memory accesses appear to a viewer *outside* the executing processor (for instance, in a multiprocessor system) is not specified by the architecture.

The SYNC instruction can be used to create a point in the executing instruction stream at which the relative order of some loads and stores can be determined: loads and stores executed before the SYNC are completed before loads and stores after the SYNC can start.

Table 4-15 lists the SYNC instruction, along with the MIPS ISA within which it is defined.

Table 4-15 Serialization Instruction

Mnemonic	Instruction	Defined in MIPS ISA
SYNC	Synchronize Shared Memory	MIPS32

4.1.4.2 Exception Instructions

Exception instructions transfer control to a software exception handler in the kernel. There are two types of exceptions, *conditional* and *unconditional*. These are caused by the following instructions:

Trap instructions, which cause conditional exceptions based upon the result of a comparison

System call and breakpoint instructions, which cause unconditional exceptions

Table 4-16 lists the system call and breakpoint instructions. Table 4-17 lists the trap instructions that compare two registers. Table 4-18 lists trap instructions, which compare a register value with an *immediate* value.

Each table also lists the MIPS ISA within which an instruction is defined.

Table 4-16 System Call and Breakpoint Instructions

Mnemonic	Instruction	Defined in MIPS ISA
BREAK	Breakpoint	MIPS32
SYSCALL	System Call	MIPS32

Table 4-17 Trap-on-Condition Instructions Comparing Two Registers

Mnemonic	Instruction	Defined in MIPS ISA
TEQ	Trap if Equal	MIPS32
TGE	Trap if Greater Than or Equal	MIPS32
TGEU	Trap if Greater Than or Equal Unsigned	MIPS32
TLT	Trap if Less Than	MIPS32
TLTU	Trap if Less Than Unsigned	MIPS32II

Table 4-17 Trap-on-Condition Instructions Comparing Two Registers

Mnemonic	Instruction	Defined in MIPS ISA
TNE	Trap if Not Equal	MIPS32

Table 4-18 Trap-on-Condition Instructions Comparing an Immediate Value

Mnemonic	Instruction	Defined in MIPS ISA
TEQI	Trap if Equal Immediate	MIPS32
TGEI	Trap if Greater Than or Equal Immediate	MIPS32
TGEIU	Trap if Greater Than or Equal Immediate Unsigned	MIPS32
TLTI	Trap if Less Than Immediate	MIPS32
TLTIU	Trap if Less Than Immediate Unsigned	MIPS32
TNEI	Trap if Not Equal Immediate	MIPS32

4.1.4.3 Conditional Move Instructions

MIPS32 includes instructions to conditionally move one CPU general register to another, based on the value in a third general register. For floating point conditional moves, refer to Chapter 4.

Table 4-19 lists conditional move instructions, along with the MIPS ISA within which an instruction is defined.

Table 4-19 CPU Conditional Move Instructions

Mnemonic	Instruction	Defined in MIPS ISA
MOVF	Move Conditional on Floating Point False	MIPS32
MOVN	Move Conditional on Not Zero	MIPS32
MOVT	Move Conditional on Floating Point True	MIPS32
MOVZ	Move Conditional on Zero	MIPS32

4.1.4.4 Prefetch Instructions

There is one prefetch advisory instruction:

- One with register+offset addressing (PREF)

These instructions advise that memory is likely to be used in a particular way in the near future and should be prefetched into the cache.

Table 4-20 Prefetch Instructions

Mnemonic	Instruction	Addressing Mode	Defined in MIPS ISA
PREF	Prefetch	Register+Offset	MIPS32

4.1.4.5 NOP Instructions

The NOP instruction is actually encoded as an all-zero instruction. MIPS processors special-case this encoding as performing no operation, and optimize execution of the instruction. In addition, SSNOP instruction, takes up one issue cycle on any processor, including super-scalar implementations of the architecture.

Table 4-21 lists conditional move instructions, along with the MIPS ISA within which an instruction is defined.

Table 4-21 NOP Instructions

Mnemonic	Instruction	Defined in MIPS ISA
NOP	No Operation	MIPS32
SSNOP	Superscalar Inhibit NOP	MIPS32

4.1.5 Coprocessor Instructions

This section contains information about the following:

- "What Coprocessors Do"
- "System Control Coprocessor 0 (CP0)"
- "Floating Point Coprocessor 1 (CP1)"
- "Coprocessor Load and Store Instructions"

4.1.5.1 What Coprocessors Do

Coprocessors are alternate execution units, with register files separate from the CPU. In abstraction, the MIPS architecture provides for up to four coprocessor units, numbered 0 to 3. Each level of the ISA defines a number of these coprocessors, as listed in Table 4-22.

Table 4-22 Coprocessor Definition and Use in the MIPS Architecture

Coprocessor	MIPS32	MIPS64
CP0	Sys Control	Sys Control
CP1	FPU	FPU
CP2	implementation specific	
CP3	implementation specific	FPU (COP1X)

Coprocessor 0 is always used for system control and coprocessor 1 and 3 are used for the floating point unit. Coprocessor 2 is reserved for implementation-specific use.

A coprocessor may have two different register sets:

- Coprocessor general registers
- Coprocessor control registers

Each set contains up to 32 registers. Coprocessor computational instructions may use the registers in either set.

4.1.5.2 System Control Coprocessor 0 (CP0)

The system controller for all MIPS processors is implemented as coprocessor 0 (CP0[1]), the **System Control Coprocessor**. It provides the processor control, memory management, and exception handling functions.

4.1.5.3 Floating Point Coprocessor 1 (CP1)

If a system includes a **Floating Point Unit**, it is implemented as coprocessor 1 (CP1[2]). Details of the FPU instructions are documented in Chapter 5, “Overview of the FPU Instruction Set,” on page 39.

Coprocessor instructions are divided into two main groups:

- Load and store instructions (move to and from coprocessor), which are reserved in the main *opcode* space
- Coprocessor-specific operations, which are defined entirely by the coprocessor

4.1.5.4 Coprocessor Load and Store Instructions

Explicit load and store instructions are not defined for CP0; for CP0 only, the move to and from coprocessor instructions must be used to write and read the CP0 registers. The loads and stores for the remaining coprocessors are summarized in “Coprocessor Loads and Stores” on page 27.

4.2 CPU Instruction Formats

A CPU instruction is a single 32-bit aligned word. The CPU instruction formats are shown below:

- Immediate (see Figure 4-1)
- Jump (see Figure 4-2)
- Register (see Figure 4-3)

[1] CP0 instructions use the COP0 opcode, and as such are differentiated from the CP0 designation in this book.

[2] FPU instructions (such as LWC1, SDC1, etc.) that use the COP1 opcode are differentiated from the CP1 designation in this book. See Chapter 5, “Overview of the FPU Instruction Set,” on page 39 for more information about the FPU instructions.

Table 4-23 describes the fields used in these instructions.

Table 4-23 CPU Instruction Format Fields

Field	Description
opcode	6-bit primary operation code
rd	5-bit specifier for the destination register
rs	5-bit specifier for the source register
rt	5-bit specifier for the target (source/destination) register or used to specify functions within the primary *opcode* REGIMM
immediate	16-bit signed *immediate* used for logical operands, arithmetic signed operands, load/store address byte offsets, and PC-relative branch signed instruction displacement
instr_index	26-bit index shifted left two bits to supply the low-order 28 bits of the jump target address
sa	5-bit shift amount
function	6-bit function field used to specify functions within the primary *opcode* SPECIAL

Figure 4-1 Immediate (I-Type) CPU Instruction Format

31 26	25 21	20 16	15 0
opcode	rs	rt	immediate
6	5	5	16

Figure 4-2 Jump (J-Type) CPU Instruction Format

31 26	25 21 20 16 15 11 10 6 5 0
opcode	instr_index
6	26

Figure 4-3 Register (R-Type) CPU Instruction Format

31 26	25 21	20 16	15 11	10 6	5 0
opcode	rs	rt	rd	sa	function
6	5	5	5	5	6

Chapter 5c

Overview of the FPU Instruction Set

This chapter describes the instruction set architecture (ISA) for the floating point unit (FPU) in the MIPS32 architecture. In the MIPS architecture, the FPU is implemented via Coprocessor 1 and Coprocessor 3, an optional processor implementing IEEE Standard 754[1] floating point operations. The FPU also provides a few additional operations not defined by the IEEE standard.

This chapter provides an overview of the following FPU architectural details:

- Section 5.1 , "Binary Compatibility"
- Section 5.2 , "Enabling the Floating Point Coprocessor"
- Section 5.3 , "IEEE Standard 754"
- Section 5.4 , "FPU Data Types"
- Section 5.5 , "Floating Point Register Types"
- Section 5.6 , "Floating Point Control Registers (FCRs)"
- Section 5.7 , "Formats of Values Used in FP Registers"
- Section 5.8 , "FPU Exceptions"
- Section 5.9 , "FPU Instructions"
- Section 5.10 , "Valid Operands for FPU Instructions"
- Section 5.11 , "FPU Instruction Formats"

The FPU instruction set is summarized by functional group. Each instruction is also described individually in alphabetical order in Volume II.

5.1 Binary Compatibility

In addition to an Instruction Set Architecture, the MIPS architecture definition includes processing resources such as the set of coprocessor general registers. The 32-bit registers in MIPS32 were enlarged to 64-bits in MIPS64; however, these 64-bit FPU registers are not backwards compatible. Instead, processors implementing the MIPS64 Architecture provide a mode bit to select either the 32-bit or 64-bit register model.

Any processor implementing MIPS64 can also run MIPS32 binary programs without change.

5.2 Enabling the Floating Point Coprocessor

Enabling the Floating Point Coprocessor is done by enabling Coprocessor 1, and is a privileged operation provided by the System Control Coprocessor. If Coprocessor 1 is not enabled, an attempt to execute a floating point instruction causes

[1] In this chapter, references to “IEEE standard” and “IEEE Standard 754” refer to IEEE Standard 754-1985, “IEEE Standard for Binary Floating Point Arithmetic.” For more information about this standard, see the IEEE web page at http://stdsbbs.ieee.org/.

a Coprocessor Unusable exception. Every system environment either enables the FPU automatically or provides a means for an application to request that it is enabled.

5.3 IEEE Standard 754

IEEE Standard 754 defines the following:

- Floating point data types
- The basic arithmetic, comparison, and conversion operations
- A computational model

The IEEE standard does not define specific processing resources nor does it define an instruction set.

The MIPS architecture includes non-IEEE FPU control and arithmetic operations (multiply-add, reciprocal, and reciprocal square root) which may not supply results that match the IEEE precision rules.

5.4 FPU Data Types

The FPU provides both floating point and fixed point data types, which are described in the next two sections.

- The single and double precision floating point data types are those specified by the IEEE standard.
- The fixed point types are signed integers provided by the CPU architecture.

5.4.1 Floating Point Formats

The following three floating point formats are provided by the FPU:

- 32-bit **single precision** floating point (type *S*, shown in Figure 5-1)
- 64-bit **double precision** floating point (type *D*, shown in Figure 5-2)

The floating point data types represent numeric values as well as other special entities, such as the following:

- Two infinities, +∞ and -∞
- Signaling non-numbers (SNaNs)
- Quiet non-numbers (QNaNs)s
- Numbers of the form: $(-1)^s \, 2^E \, b_0.b_1 \, b_2..b_{p-1}$, where:
  - s=0 or 1
  - E=any integer between *E_min* and *E_max*, inclusive
  - b_i=0 or 1 (the high bit, b_0, is to the left of the binary point)
  - p is the signed-magnitude precision

Table 5-1 Parameters of Floating Point Data Types

Parameter	Single	Double
Bits of mantissa precision, p	24	53
Maximum exponent, E_max	+127	+1023

Table 5-1 Parameters of Floating Point Data Types

Parameter	Single	Double
Minimum exponent, E_min	-126	-1022
Exponent *bias*	+127	+1023
Bits in exponent field, *e*	8	11
Representation of b_0 integer bit	hidden	hidden
Bits in fraction field, *f*	23	52
Total format width in bits	32	64

The single and double floating point data types are composed of three fields—*sign*, *exponent*, *fraction*—whose sizes are listed in Table 5-1.

Layouts of these fields are shown in Figures 5-1, and 5-2 below. The fields are

- 1-bit sign, *s*
- Biased exponent, $e = E + bias$
- Binary fraction, $f = .b_1\ b_2..b_{p-1}$ (the b_0 bit is not recorded)

Figure 5-1 Single-Precisions Floating Point Format (S)

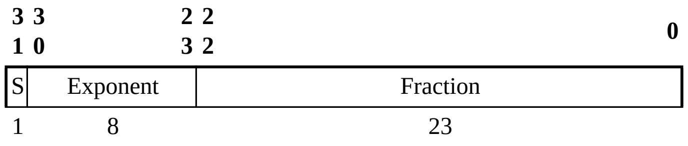
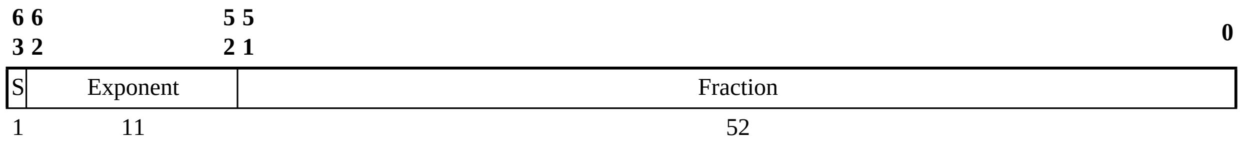


Figure 5-2 Double-Precisions Floating Point Format (D)



Values are encoded in the specified format by using unbiased exponent, fraction, and sign values listed in Table 5-2. The high-order bit of the *Fraction* field, identified as b_1, is also important for NaNs.

Table 5-2 Value of Single or Double Floating Point DataType Encoding

Unbiased E	f	s	b_1	Value V	Type of Value	Typical Single Bit Pattern[a]	Typical Double Bit Pattern[a.]
E_max + 1	≠ 0		1	SNaN	Signaling NaN	`16#7fffffff`	`16#7fffffff ffffffff`
			0	QNaN	Quiet NaN	`16#7fbfffff`	`16#7ff7ffff ffffffff`
E_max +1	0	1		- ∞	minus infinity	`16#ff800000`	`16#fff00000 00000000`
		0		+ ∞	plus infinity	`16#7f800000`	`16#7ff00000 00000000`

Table 5-2 Value of Single or Double Floating Point DataType Encoding

Unbiased E	f	s	b_1	Value V	Type of Value	Typical Single Bit Pattern[a]	Typical Double Bit Pattern[a]
E_max to *E_min*		1		$-\ (2^{E})(1.f)$	negative normalized number	`16#80800000` through `16#ff7fffff`	`16#80100000 00000000` through `16#ffefffff ffffffff`
		0		$+\ (2^{E})(1.f)$	positive normalized number	`16#00800000` through `16#7f7fffff`	`16#00100000 00000000` through `16#7fefffff ffffffff`
E_min -1	≠ 0	1		$-\ (2^{E\_min})(0.f)$	negative denormalized number	`16#807fffff`	`16#800fffff ffffffff`
		0		$+\ (2^{E\_min})(0.f)$	positive denormalized number	`16#007fffff`	`16#00ffffff ffffffff`
E_min -1	0	1		- 0	negative zero	`16#80000000`	`16#80000000 00000000`
		0		+ 0	positive zero	`16#00000000`	`16#00000000 00000000`

a. The "Typical" nature of the bit patterns for the NaN and denormalized values reflects the fact that the sign may have either value (NaN) and the fact that the fraction field may have any non-zero value (both). As such, the bit patterns shown are one value in a class of potential values that represent these special values.

5.4.1.1 Normalized and Denormalized Numbers

For single and double data types, each representable nonzero numerical value has just one encoding; numbers are kept in normalized form. The high-order bit of the *p*-bit mantissa, which lies to the left of the binary point, is “hidden,” and not recorded in the *Fraction* field. The encoding rules permit the value of this bit to be determined by looking at the value of the exponent. When the unbiased exponent is in the range *E_min* to *E_max*, inclusive, the number is normalized and the hidden bit must be 1. If the numeric value cannot be normalized because the exponent would be less than *E_min*, then the representation is denormalized and the encoded number has an exponent of *E_min*-1 and the hidden bit has the value 0. Plus and minus zero are special cases that are not regarded as denormalized values.

5.4.1.2 Reserved Operand Values—Infinity and NaN

A floating point operation can signal IEEE exception conditions, such as those caused by uninitialized variables, violations of mathematical rules, or results that cannot be represented. If a program does not choose to trap IEEE exception conditions, a computation that encounters these conditions proceeds without trapping but generates a result indicating that an exceptional condition arose during the computation. To permit this, each floating point format defines representations, listed in Table 5-2, for plus infinity (+∞), minus infinity (-∞), quiet non-numbers (QNaN), and signaling non-numbers (SNaN).

5.4.1.3 Infinity and Beyond

Infinity represents a number with magnitude too large to be represented in the format; in essence it exists to represent a magnitude overflow during a computation. A correctly signed ∞ is generated as the default result in division by zero and some cases of overflow; details are given in the IEEE exception condition described in.

Once created as a default result, ∞ can become an operand in a subsequent operation. The infinities are interpreted such that -∞ < (every finite number) < +∞. Arithmetic with ∞ is the limiting case of real arithmetic with operands of arbitrarily large magnitude, when such limits exist. In these cases, arithmetic on ∞ is regarded as exact and exception conditions do not arise. The out-of-range indication represented by ∞ is propagated through subsequent computations. For some cases there is no meaningful limiting case in real arithmetic for operands of ∞, and these cases raise the Invalid Operation exception condition (see “Invalid Operation Exception” on page 53).

5.4.1.4 Signalling Non-Number (SNaN)

SNaN operands cause the Invalid Operation exception for arithmetic operations. SNaNs are useful values to put in uninitialized variables. An SNaN is never produced as a result value.

IEEE Standard 754 states that “Whether copying a signaling NaN without a change of format signals the Invalid Operation exception is the implementor’s option.” The MIPS architecture has chosen to make the formatted operand move instructions (MOV.fmt MOVT.fmt MOVF.fmt MOVN.fmt MOVZ.fmt) non-arithmetic and they do not signal IEEE 754 exceptions.

5.4.1.5 Quiet Non-Number (QNaN)

QNaNs are intended to afford retrospective diagnostic information inherited from invalid or unavailable data and results. Propagation of the diagnostic information requires information contained in a QNaN to be preserved through arithmetic operations and floating point format conversions.

QNaN operands do not cause arithmetic operations to signal an exception. When a floating point result is to be delivered, a QNaN operand causes an arithmetic operation to supply a QNaN result. When possible, this QNaN result is one of the operand QNaN values. QNaNs do have effects similar to SNaNs on operations that do not deliver a floating point result—specifically, comparisons. (For more information, see the detailed description of the floating point compare instruction, C.cond.fmt.)

When certain invalid operations not involving QNaN operands are performed but do not trap (because the trap is not enabled), a new QNaN value is created. Table 5-3 shows the QNaN value generated when no input operand QNaN value can be copied. The values listed for the fixed point formats are the values supplied to satisfy the IEEE standard when a QNaN or infinite floating point value is converted to fixed point. There is no other feature of the architecture that detects or makes use of these “integer QNaN” values.

Table 5-3 Value Supplied When a New Quiet NaN Is Created

Format	New QNaN value
Single floating point	`16#7fbf ffff`
Double floating point	`16#7ff7 ffff ffff ffff`
Word fixed point	`16#7fff ffff`

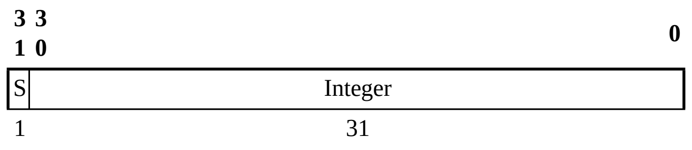
Fixed Point Formats

The FPU provides one fixed point data type:

- 32-bit **Word** fixed point (type *W*), shown in Figure 5-3

The fixed point values are held in the 2’s complement format used for signed integers in the CPU. Unsigned fixed point data types are not provided by the architecture; application software may synthesize computations for unsigned integers from the existing instructions and data types.

Figure 5-3 Word Fixed Point Format (W)



5.5 Floating Point Register Types

This section describes the organization and use of the two types of FPU register sets:

- *Floating Point* registers (*FPR*s) are 64 bits wide. Depending on the mode of operation, there are either 16 or 32 FPR registers in the register file. The FR Bit of the CP0 Status register determines which mode is selected:
  - When **The FR Bit** is a 1, the FPU defines 32 FPRs
  - When **The FR Bit** is a 0, the FPU defines 16 FPRs (this mode is supported only for backward compatibility with the MIPS32 Architecture)

These registers transfer binary data between the FPU and the system, and are also used to hold formatted FPU operand values. Refer to Volume III, The MIPS Privileged Architecture Manual, for more information on the CP0 Registers.

- *Floating Point Control* registers (*FCR*s), which are 32 bits wide. There are five FPU control registers, used to identify and control the FPU. These registers are indicated by the *fs* field of the instruction word. Three of these registers, *FCCR, FEXR,* and *FENR,* select subsets of the floating point *Control/Status* register, the *FCSR*.

5.5.1 FPRs and Formatted Operand Layout

FPU instructions that operate on formatted operand values specify the **floating point register** (FPR) that holds the value.

5.6 Floating Point Control Registers (FCRs)

The MIPS32 Architecture supports the following five floating point *Control* registers (*FCRs*):

- *FIR*, FP *Implementation and Revision* register
- *FCCR*, FP *Condition Codes* register
- *FEXR*, FP *Exceptions* register
- *FENR*, FP *Enables* register
- *FCSR*, FP *Control/Status* register (used to be known as *FCR31*).

FCCR, *FEXR*, and *FENR* access portions of the *FCSR* through CTC1 and CFC1 instructions.

Access to the Floating Point Control Registers is not privileged; they can be accessed by any program that can execute floating point instructions. The FCRs can be accessed via the CTC1 and CFC1 instructions.

5.6.1 Floating Point Implementation Register (FCCR, CP1 Control Register 0)

Compliance Level: *Required* if floating point is implemented

The Floating Point Implementation Register (*FIR*) is a 32-bit read-only register that contains information identifying the capabilities of the floating point unit, the floating point processor identification, and the revision level of the floating point unit. Figure 5-4 shows the format of the *FIR* register; Table 5-4 describes the *FIR* register fields.

Figure 5-4 FIR Register Format

31 20	19	18	17	16	15 8	7 0
0 0000 0000 0000	3D	PS	D	S	ProcessorID	Revision

Table 5-4 FIR Register Field Descriptions

Fields		Description	Read/ Write	Reset State	Compliance
Name	**Bits**				
0	31:20	Reserved for future use; reads as zero	0	0	Reserved
3D	19	Used by MIPS64 processors to indicate that the MIPS-3D ASE is implemented. Not used by MIPS32 processors and always reads as zero.	0	0	Required
PS	18	Used by MIPS64 processors to indicate that the pair-single floating point data type is implemented. Not used by MIPS32 processors and always reads as zero.	0	0	Required
D	17	Indicates that the double-precision (D) floating point data type and instructions are implemented: 0: D floating not implemented 1: D floating implemented	R	Preset	Required
S	16	Indicates that the single-precision (S) floating point data type and instructions are implemented: 0: S floating not implemented 1: S floating implemented	R	Preset	Required
ProcessorID	15:8	Identifies the floating point processor.	R	Preset	Required
Revision	7:0	Specifies the revision number of the floating point unit. This field allows software to distinguish between one revision and another of the same floating point processor type. If this field is not implemented, it must read as zero.	R	Preset	Optional

5.6.2 Floating Point Control and Status Register (FCSR, CP1 Control Register 31)

Compliance Level: *Required* if floating point is implemented.

The Floating Point Control and Status Register (*FCSR*) is a 32-bit register that controls the operation of the floating point unit, and shows the following status information:

- selects the default rounding mode for FPU arithmetic operations
- selectively enables traps of FPU exception conditions
- controls some denormalized number handling options
- reports any IEEE exceptions that arose during the most recently executed instruction
- reports IEEE exceptions that arose, cumulatively, in completed instructions
- indicates the condition code result of FP compare instructions

Access to *FCSR* is not privileged; it can be read or written by any program that has access to the floating point unit (via the coprocessor enables in the *Status* register). Figure 5-5 shows the format of the *FCSR* register; Table 5-5 describes the *FCSR* register fields.

Figure 5-5 FCSR Register Format

31	30	29	28	27	26	25	24	23	22	21	20		18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
FCC							FS	FCC	Impl		0 000			Cause						Enables					Flags					RM	
7	6	5	4	3	2	1		0						E	V	Z	O	U	I	V	Z	O	U	I	V	Z	O	U	I		

Table 5-5 FCSR Register Field Descriptions

Fields		Description	Read/ Write	Reset State	Compliance
Name	Bits				
FCC	31:25, 23	Floating point condition codes. These bits record the result of floating point compares and are tested for floating point conditional branches and conditional moves. The FCC bit to use is specified in the compare, branch, or conditional move instruction. For backward compatibility with previous MIPS ISAs, the FCC bits are separated into two, non-contiguous fields.	R/W	Undefined	Required
FS	24	Flush to Zero. When FS is one, denormalized results are flushed to zero instead of causing an Unimplemented Operation exception. It is implementation dependent whether denormalized operand values are flushed to zero before the operation is carried out.	R/W	Undefined	Required
Impl	22:21	Available to control implementation dependent features of the floating point unit. If these bits are not implemented, they must be ignored on write and read as zero.	R/W	Undefined	Optional
0	20:18	Reserved for future use; Must be written as zero; returns zero on read.	0	0	Reserved
Cause	17:12	Cause bits. These bits indicate the exception conditions that arise during execution of an FPU arithmetic instruction. A bit is set to 1 if the corresponding exception condition arises during the execution of an instruction and is set to 0 otherwise. By reading the registers, the exception condition caused by the preceding FPU arithmetic instruction can be determined. Refer to Table 5-6 for the meaning of each bit.	R/W	Undefined	Required
Enables	11:7	Enable bits. These bits control whether or not a exception is taken when an IEEE exception condition occurs for any of the five conditions. The exception occurs when both an Enable bit and the corresponding Cause bit are set either during an FPU arithmetic operation or by moving a value to FCSR or one of its alternative representations. Note that Cause bit E has no corresponding Enable bit; the non-IEEE Unimplemented Operation exception is defined by MIPS as always enabled. Refer to Table 5-6 for the meaning of each bit.	R/W	Undefined	Required

Table 5-5 FCSR Register Field Descriptions

Fields		Description	Read/ Write	Reset State	Compliance
Name	Bits				
Flags	6:2	Flag bits. This field shows any exception conditions that have occurred for completed instructions since the flag was last reset by software. When a FPU arithmetic operation raises an IEEE exception condition that does not result in a Floating Point Exception (i.e., the Enable bit was off), the corresponding bit(s) in the Flag field are set, while the others remain unchanged. Arithmetic operations that result in a Floating Point Exception (i.e., the Enable bit was on) do not update the Flag bits. This field is never reset by hardware and must be explicitly reset by software. Refer to Table 5-6 for the meaning of each bit.	R/W	Undefined	Required
RM	1:0	Rounding mode. This field indicates the rounding mode used for most floating point operations (some operations use a specific rounding mode). Refer to Table 5-7 for the meaning of the encodings of this field.	R/W	Undefined	Required.

The FCC, FS, Cause, Enables, Flags and RM fields in the FCSR, FCCR, FEXR, and FENR registers always display the correct state. That is, if a field is written via FCCR, the new value may be read via one of the alternate registers. Similarly, if a value is written via one of the alternate registers, the new value may be read via FCSR.

Table 5-6 Cause, Enable, and Flag Bit Definitions

Bit Name	Bit Meaning
E	Unimplemented Operation (this bit exists only in the Cause field)
V	Invalid Operation
Z	Divide by Zero
O	Overflow
U	Underflow
I	Inexact

Table 5-7 Rounding Mode Definitions

RM Field Encoding	Meaning
0	RN - Round to Nearest Rounds the result to the nearest representable value. When two representable values are equally near, the result is rounded to the value whose least significant bit is zero (that is, even)
1	RZ - Round Toward Zero Rounds the result to the value closest to but not greater than in magnitude than the result.

Table 5-7 Rounding Mode Definitions

RM Field Encoding	Meaning
2	RP - Round Towards Plus Infinity Rounds the result to the value closest to but not less than the result.
3	RM - Round Towards Minus Infinity Rounds the result to the value closest to but not greater than the result.

5.6.3 Floating Point Condition Codes Register (FCCR, CP1 Control Register 25)

Compliance Level: *Required* if floating point is implemented.

The Floating Point Condition Codes Register (*FCCR*) is an alternative way to read and write the floating point condition code values that also appear in *FCSR*. Unlike *FCSR*, all eight FCC bits are contiguous in *FCCR*. Figure 5-6 shows the format of the *FCCR* register; Table 5-8 describes the *FCCR* register fields.

Figure 5-6 FCCR Register Format

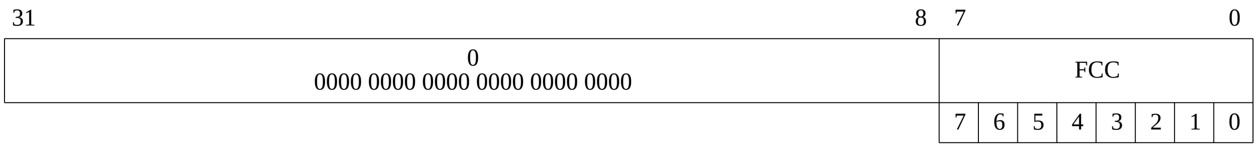


Table 5-8 FCCR Register Field Descriptions

Fields		Description	Read/ Write	Reset State	Compliance
Name	Bits				
0	31:8	Must be written as zero; returns zero on read	0	0	Reserved
FCC	7:0	Floating point condition code. Refer to the description of this field in the *FCSR* register.	R/W	Undefined	Required

5.6.4 Floating Point Exceptions Register (FEXR, CP1 Control Register 26)

Compliance Level: *Required* if floating point is implemented.

The Floating Point Exceptions Register (*FEXR*) is an alternative way to read and write the Cause and Flags fields that also appear in *FCSR*. Figure 5-7 shows the format of the *FEXR* register; Table 5-9 describes the *FEXR* register fields.

Figure 5-7 FEXR Register Format

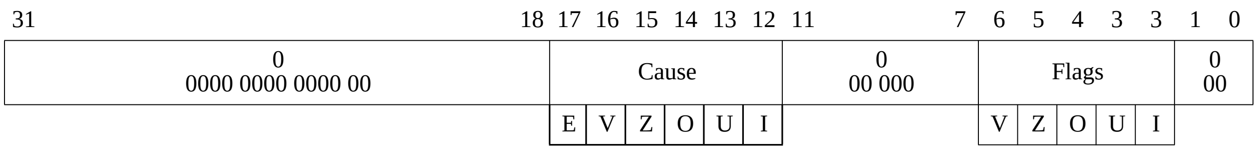


Table 5-9 FEXR Register Field Descriptions

Fields		Description	Read/ Write	Reset State	Compliance
Name	Bits				
0	31:18, 11:7, 1:0	Must be written as zero; returns zero on read	0	0	Reserved
Cause	17:12	Cause bits. Refer to the description of this field in the *FCSR* register.	R/W	Undefined	Required
Flags	6:2	Flags bits. Refer to the description of this field in the *FCSR* register.	R/W	Undefined	Optional

5.6.5 Floating Point Enables Register (FENR, CP1 Control Register 28)

Compliance Level: *Required* if floating point is implemented.

The Floating Point Enables Register (*FENR*) is an alternative way to read and write the Enables, FS, and RM fields that also appear in *FCSR*. Figure 5-8 shows the format of the *FENR* register; Table 5-10 describes the *FENR* register fields.

Figure 5-8 FENR Register Format

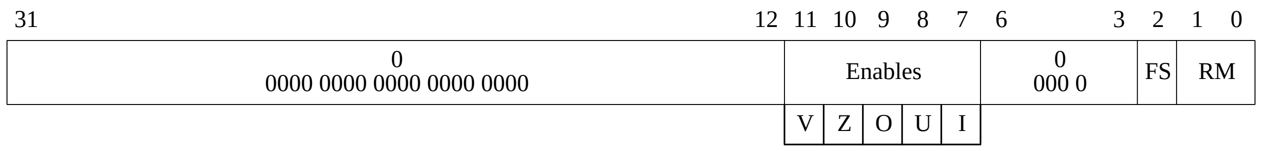


Table 5-10 FENR Register Field Descriptions

Fields		Description	Read/ Write	Reset State	Compliance
Name	Bits				
0	31:12, 6:3	Must be written as zero; returns zero on read	0	0	Reserved
Enables	11:7	Enable bits. Refer to the description of this field in the *FCSR* register.	R/W	Undefined	Required
FS	2	Flush to Zero bit. Refer to the description of this field in the *FCSR* register.	R/W	Undefined	Required
RM	1:0	Rounding mode. Refer to the description of this field in the *FCSR* register.	R/W	Undefined	Required

5.7 Formats of Values Used in FP Registers

Unlike the CPU, the FPU does not interpret the binary encoding of source operands nor produce a binary encoding of results for every operation. The value held in a floating point operand register (FPR) has a format, or type, and it may be used only by instructions that operate on that format. The format of a value is either *uninterpreted*, *unknown*, or one of the valid numeric formats: *single* and *double* floating point, and *word* and *long* fixed point.

The value in an FPR is always set when a value is written to the register:

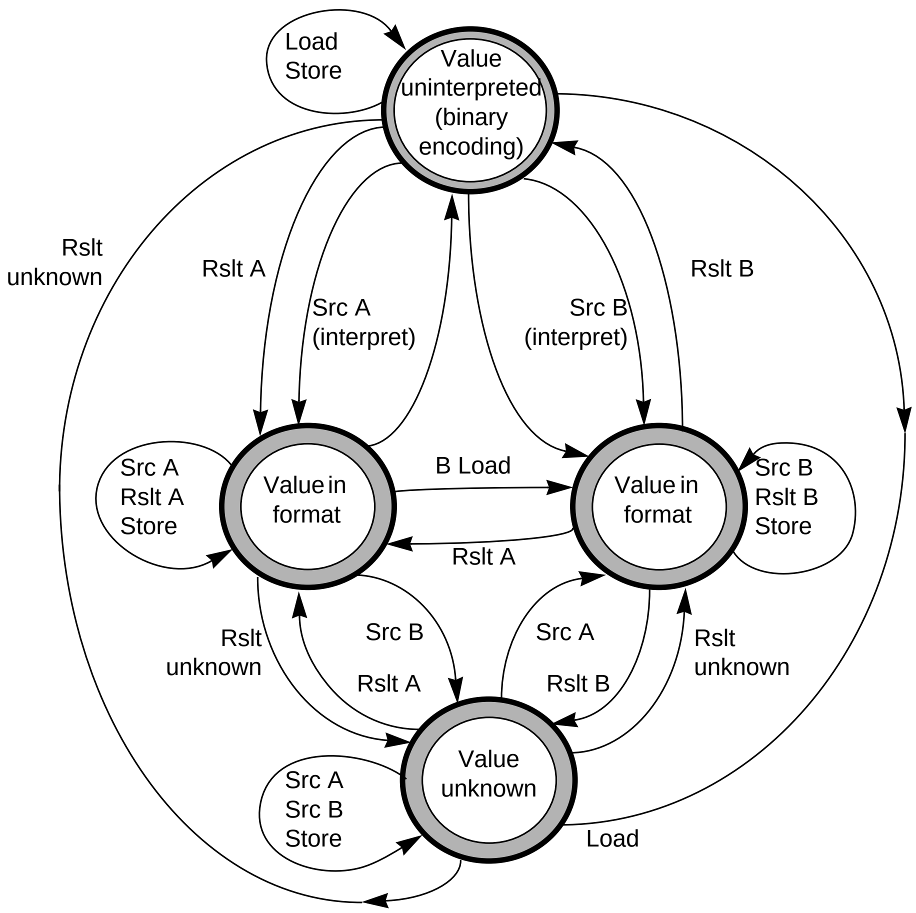
- When a data transfer instruction writes binary data into an FPR (a load), the FPR receives a binary value that is *uninterpreted*.
- A computational or FP register move instruction that produces a result of type *fmt* puts a value of type *fmt* into the result register.

When an FPR with an *uninterpreted* value is used as a source operand by an instruction that requires a value of format *fmt*, the binary contents are interpreted as an encoded value in format *fmt* and the value in the FPR changes to a value of format *fmt*. The binary contents cannot be reinterpreted in a different format.

If an FPR contains a value of format *fmt*, a computational instruction must not use the FPR as a source operand of a different format. If this occurs, the value in the register becomes *unknown* and the result of the instruction is also a value that is *unknown*. Using an FPR containing an *unknown* value as a source operand produces a result that has an *unknown* value.

The format of the value in the FPR is unchanged when it is read by a data transfer instruction (a store). A data transfer instruction produces a binary encoding of the value contained in the FPR. If the value in the FPR is *unknown*, the encoded binary value produced by the operation is not defined.

The state diagram in Figure 5-9 illustrates the manner in which the formatted value in an FPR is set and changed.



A, B:Example formats
Load:Destination of LWC1, LDC1, or MTC1 instructions.
Store:Source operand of SWC1, SDC1, or MFC1 instructions.
Src fmt:Source operand of computational instruction expecting format “fmt.”
Rslt fmt:Result of computational instruction producing value of format “fmt.”

Figure 5-9 Effect of FPU Operations on the Format of Values Held in FPRs

5.8 FPU Exceptions

This section provides the following information FPU exceptions:

- Precise exception mode
- Descriptions of the exceptions

FPU exceptions are implemented in the MIPS FPU architecture with the *Cause*, *Enable*, and *Flag* fields of the *Control/Status* register. The *Flag* bits implement IEEE exception status flags, and the *Cause* and *Enable* bits control exception trapping. Each field has a bit for each of the five IEEE exception conditions and the *Cause* field has an additional exception bit, Unimplemented Operation, used to trap for software emulation assistance.

5.8.0.1 Precise Exception Mode

In precise exception mode, a trap occurs before the instruction that causes the trap, or any following instruction, can complete and write its results. If desired, the software trap handler can resume execution of the interrupted instruction stream after handling the exception.

The *Cause* field reports per-bit instruction exception conditions. The *Cause* bits are written during each floating point arithmetic operation to show any exception conditions that arise during the operation. The bit is set to 1 if the corresponding exception condition arises; otherwise it is set to 0.

A floating point trap is generated any time both a *Cause* bit and its corresponding *Enable* bit are set. This occurs either during the execution of a floating point operation or by moving a value into the *FCSR*. There is no *Enable* for Unimplemented Operation; this exception always generates a trap.

In a trap handler, exception conditions that arise during any trapped floating point operations are reported in the *Cause* field. Before returning from a floating point interrupt or exception, or before setting *Cause* bits with a move to the *FCSR*, software must first clear the enabled *Cause* bits by executing a move to *FCSR* to prevent the trap from being erroneously retaken.

User-mode programs cannot observe enabled *Cause* bits being set. If this information is required in a User-mode handler, it must be available someplace other than through the *Status* register.

If a floating point operation sets only non-enabled *Cause* bits, no trap occurs and the default result defined by the IEEE standard is stored (see Table 5-11). When a floating point operation does not trap, the program can monitor the exception conditions by reading the *Cause* field.

The *Flag* field is a cumulative report of IEEE exception conditions that arise as instructions complete; instructions that trap do not update the *Flag* bits. The *Flag* bits are set to 1 if the corresponding IEEE exception is raised, otherwise the bits are unchanged. There is no *Flag* bit for the MIPS Unimplemented Operation exception. The *Flag* bits are never cleared as a side effect of floating point operations, but may be set or cleared by moving a new value into the *FCSR*.

Addressing exceptions are precise.

5.8.1 Exception Conditions

The following five exception conditions defined by the IEEE standard are described in this section:

- “Invalid Operation Exception”
- “Division By Zero Exception”
- “Underflow Exception”
- “Overflow Exception”
- “Inexact Exception”

This section also describes a MIPS-specific exception condition, **Unimplemented Operation**, that is used to signal a need for software emulation of an instruction. Normally an IEEE arithmetic operation can cause only one exception condition; the only case in which two exceptions can occur at the same time are Inexact With Overflow and Inexact With Underflow.

At the program’s direction, an IEEE exception condition can either cause a trap or not cause a trap. The IEEE standard specifies the result to be delivered in case the exception is not enabled and no trap is taken. The MIPS architecture supplies these results whenever the exception condition does not result in a precise trap (that is, no trap or an imprecise trap). The default action taken depends on the type of exception condition, and in the case of the Overflow, the current rounding mode. The default results are summarized in Table 5-11.

Table 5-11 Default Result for IEEE Exceptions Not Trapped Precisely

Bit	Description	Default Action
V	Invalid Operation	Supplies a quiet NaN.
Z	Divide by zero	Supplies a properly signed infinity.
U	Underflow	Supplies a rounded result.
I	Inexact	Supplies a rounded result. If caused by an overflow without the overflow trap enabled, supplies the overflowed result.
O	Overflow	Depends on the rounding mode, as shown below.
	0 (RN)	Supplies an infinity with the sign of the intermediate result.
	1 (RZ)	Supplies the format’s largest finite number with the sign of the intermediate result.
	2 (RP)	For positive overflow values, supplies positive infinity. For negative overflow values, supplies the format’s most negative finite number.
	3 (RM)	For positive overflow values, supplies the format’s largest finite number. For negative overflow values, supplies minus infinity.

5.8.1.1 Invalid Operation Exception

The Invalid Operation exception is signaled if one or both of the operands are invalid for the operation to be performed. The result, when the exception condition occurs without a precise trap, is a quiet NaN.

These are invalid operations:

- One or both operands are a signaling NaN (except for the non-arithmetic MOV.fmt, MOVT.fmt, MOVF.fmt, MOVN.fmt, and MOVZ.fmt instructions).
- Addition or subtraction: magnitude subtraction of infinities, such as (+∞) + (-∞) or (-∞) - (-∞).
- Multiplication: 0 × ∞, with any signs.
- Division: 0/0 or ∞/∞, with any signs.
- Square root: An operand of less than 0 (-0 is a valid operand value).
- Conversion of a floating point number to a fixed point format when either an overflow or an operand value of infinity or NaN precludes a faithful representation in that format.
- Some comparison operations in which one or both of the operands is a QNaN value. (The detailed definition of the compare instruction, C.cond.fmt, in Volume II has tables showing the comparisons that do and do not signal the exception.)

5.8.1.2 Division By Zero Exception

An implemented divide operation signals a Division By Zero exception if the divisor is zero and the dividend is a finite nonzero number. The result, when no precise trap occurs, is a correctly signed infinity. Divisions (0/0) and (∞/0) do not cause the Division By Zero exception. The result of (0/0) is an Invalid Operation exception. The result of (∞/0) is a correctly signed infinity.

5.8.1.3 Underflow Exception

Two related events contribute to underflow:

- Tininess: the creation of a tiny nonzero result between $\pm 2^{E\_min}$ which, because it is tiny, may cause some other exception later such as overflow on division
- Loss of accuracy: the extraordinary loss of accuracy during the approximation of such tiny numbers by denormalized numbers

Tininess: The IEEE standard allows choices in detecting these events, but requires that they be detected in the same manner for all operations. The IEEE standard specifies that “tininess” may be detected at either of these times:

- *After rounding*, when a nonzero result computed as though the exponent range were unbounded would lie strictly between $\pm 2^{E\_min}$
- *Before rounding*, when a nonzero result computed as though both the exponent range and the precision were unbounded would lie strictly between $\pm 2^{E\_min}$

The MIPS architecture specifies that tininess be detected after rounding.

Loss of Accuracy: The IEEE standard specifies that loss of accuracy may be detected as a result of either of these conditions:

- *Denormalization loss*, when the delivered result differs from what would have been computed if the exponent range were unbounded
- *Inexact result*, when the delivered result differs from what would have been computed if both the exponent range and precision were unbounded

The MIPS architecture specifies that loss of accuracy is detected as inexact result.

Signalling an Underflow: When an underflow trap is not enabled, underflow is signaled only when both tininess and loss of accuracy have been detected. The delivered result might be zero, denormalized, or 2^{E_min}.

When an underflow trap is enabled (through the *FCSR Enable* field bit), underflow is signaled when tininess is detected regardless of loss of accuracy.

5.8.1.4 Overflow Exception

An Overflow exception is signaled when the magnitude of a rounded floating point result, were the exponent range unbounded, is larger than the destination format’s largest finite number.

When no precise trap occurs, the result is determined by the rounding mode and the sign of the intermediate result.

5.8.1.5 Inexact Exception

An Inexact exception is signaled if one of the following occurs:

- The rounded result of an operation is not exact
- The rounded result of an operation overflows without an overflow trap

5.8.1.6 Unimplemented Operation Exception

The Unimplemented Operation exception is a MIPS defined exception that provides software emulation support. This exception is not IEEE-compliant.

The MIPS architecture is designed so that a combination of hardware and software may be used to implement the architecture. Operations that are not fully supported in hardware cause an Unimplemented Operation exception so that software may perform the operation.

There is no *Enable* bit for this condition; it always causes a trap. After the appropriate emulation or other operation is done in a software exception handler, the original instruction stream can be continued.

5.9 FPU Instructions

The FPU instructions comprise the following functional groups:

- “Data Transfer Instructions”
- “Arithmetic Instructions”
- “Conversion Instructions”
- “Formatted Operand-Value Move Instructions”
- “Conditional Branch Instructions”
- “Miscellaneous Instructions”

5.9.1 Data Transfer Instructions

The FPU has two separate register sets: coprocessor general registers and coprocessor control registers. The FPU has a load/store architecture; all computations are done on data held in coprocessor general registers. The control registers are used to control FPU operation. Data is transferred between registers and the rest of the system with dedicated load, store, and move instructions. The transferred data is treated as unformatted binary data; no format conversions are performed, and therefore no IEEE floating point exceptions can occur.

The supported transfer operations are listed in Table 5-12.

Table 5-12 FPU Data Transfer Instructions

Transfer Direction			Data Transferred
FPU general reg	↔	Memory	Word/doubleword load/store
FPU general reg	↔	CPU general reg	Word move
FPU control reg	↔	CPU general reg	Word move

5.9.1.1 Data Alignment in Loads, Stores, and Moves

All coprocessor loads and stores operate on naturally-aligned data items. An attempt to load or store to an address that is not naturally aligned for the data item causes an Address Error exception. Regardless of byte-ordering (the endianness), the address of a word or doubleword is the smallest byte address in the object. For a big-endian machine, this is the most-significant byte; for a little-endian machine, this is the least-significant byte (endianness is described in “Byte Ordering and Endianness” on page 17).

5.9.1.2 Addressing Used in Data Transfer Instructions

The FPU has loads and stores using the same *register+offset* addressing as that used by the CPU.

Tables 5-13 through 5-14 list the FPU data transfer instructions.

Table 5-13 FPU Loads and Stores Using Register+Offset Address Mode

Mnemonic	Instruction	Defined in MIPS ISA
LDC1	Load Doubleword to Floating Point	MIPS32

Table 5-13 FPU Loads and Stores Using Register+Offset Address Mode

Mnemonic	Instruction	Defined in MIPS ISA
LWC1	Load Word to Floating Point	MIPS32
SDC1	Store Doubleword to Floating Point	MIPS32
SWC1	Store Word to Floating Point	MIPS32

Table 5-14 FPU Move To and From Instructions

Mnemonic	Instruction	Defined in MIPS ISA
CFC1	Move Control Word From Floating Point	MIPS32
CTC1	Move Control Word To Floating Point	MIPS32
MFC1	Move Word From Floating Point	MIPS32
MTC1	Move Word To Floating Point	MIPS32

5.9.2 Arithmetic Instructions

Arithmetic instructions operate on formatted data values. The results of most floating point arithmetic operations meet the IEEE standard specification for accuracy—a result is identical to an infinite-precision result that has been rounded to the specified format, using the current rounding mode. The rounded result differs from the exact result by less than one unit in the least-significant place (ULP).

FPU IEEE-approximate arithmetic operations are listed in Table 5-15.

Table 5-15 FPU IEEE Arithmetic Operations

Mnemonic	Instruction	Defined in MIPS ISA
ABS.fmt	Floating Point Absolute Value	MIPS32
ADD.fmt	Floating Point Add	MIPS32
C.cond.fmt	Floating Point Compare	MIPS32
DIV.fmt	Floating Point Divide	MIPS32
MUL.fmt	Floating Point Multiply	MIPS32
NEG.fmt	Floating Point Negate	MIPS32
SQRT.fmt	Floating Point Square Root	MIPS32
SUB.fmt	Floating Point Subtract	MIPS32

5.9.3 Conversion Instructions

These instructions perform conversions between floating point and fixed point data types. Each instruction converts values from a number of operand formats to a particular result format. Some conversion instructions use the rounding mode specified in the *Floating Control/Status* register (*FCSR*), while others specify the rounding mode directly. Table 5-16 and Table 5-17 list the FPU conversion instructions according to their rounding mode.

Table 5-16 FPU Conversion Operations Using the *FCSR* Rounding Mode

Mnemonic	Instruction	Defined in MIPS ISA
CVT.D.fmt	Floating Point Convert to Double Floating Point	MIPS32
CVT.S.fmt	Floating Point Convert to Single Floating Point	MIPS32
CVT.W.fmt	Floating Point Convert to Word Fixed Point	MIPS32

Table 5-17 FPU Conversion Operations Using a Directed Rounding Mode

Mnemonic	Instruction	Defined in MIPS ISA
CEIL.W.fmt	Floating Point Ceiling to Word Fixed Point	MIPS32
FLOOR.W.fmt	Floating Point Floor to Word Fixed Point	MIPS32
ROUND.W.fmt	Floating Point Round to Word Fixed Point	MIPS32
TRUNC.W.fmt	Floating Point Truncate to Word Fixed Point	MIPS32

5.9.4 Formatted Operand-Value Move Instructions

These instructions all move formatted operand values among FPU general registers. A particular operand type must be moved by the instruction that handles that type. There are three kinds of move instructions:

- Unconditional move
- Conditional move that tests an FPU true/false condition code
- Conditional move that tests a CPU general-purpose register against zero

Conditional move instructions operate in a way that may be unexpected. They always force the value in the destination register to become a value of the format specified in the instruction. If the destination register does not contain an operand of the specified format before the conditional move is executed, the contents become undefined. (For more information, see the individual descriptions of the conditional move instructions in Volume II.)

These instructions are listed in Tables Table 5-18 through Table 5-20.

Table 5-18 FPU Formatted Operand Move Instructions

Mnemonic	Instruction	Defined in MIPS ISA
MOV.fmt	Floating Point Move	MIPS32

Table 5-19 FPU Conditional Move on True/False Instructions

Mnemonic	Instruction	Defined in MIPS ISA
MOVF.fmt	Floating Point Move Conditional on FP False	MIPS32
MOVT.fmt	Floating Point Move Conditional on FP True	MIPS32

Table 5-20 FPU Conditional Move on Zero/Nonzero Instructions

Mnemonic	Instruction	Defined in MIPS ISA
MOVN.fmt	Floating Point Move Conditional on Nonzero	MIPS32
MOVZ.fmt	Floating Point Move Conditional on Zero	MIPS32

5.9.5 Conditional Branch Instructions

The FPU has PC-relative conditional branch instructions that test condition codes set by FPU compare instructions (C.cond.fmt).

All branches have an architectural delay of one instruction. When a branch is taken, the instruction immediately following the branch instruction is said to be in the **branch delay slot**, and it is executed before the branch to the target instruction takes place. Conditional branches come in two versions, depending upon how they handle an instruction in the delay slot when the branch is not taken and execution falls through:

- **Branch** instructions execute the instruction in the delay slot.
- **Branch likely** instructions do not execute the instruction in the delay slot if the branch is not taken (they are said to *nullify* the instruction in the delay slot).

  Although the Branch Likely instructions are included in this specification, software is strongly encouraged to avoid the use of the Branch Likely instructions, as they will be removed from a future revision of the MIPS Architecture.

The MIPS32 Architecture defines eight condition codes for use in compare and branch instructions. For backward compatibility with previous revision of the ISA, condition code bit 0 and condition code bits 1 thru 7 are in discontiguous fields in *FCSR*.

Table 5-21 lists the conditional branch (branch and branch likely) FPU instructions; Table 5-22 lists the deprecated conditional branch likely instructions.

Table 5-21 FPU Conditional Branch Instructions

Mnemonic	Instruction	Defined in MIPS ISA
BC1F	Branch on FP False	MIPS32
BC1T	Branch on FP True	MIPS32

Table 5-22 Deprecated FPU Conditional Branch Likely Instructions

Mnemonic	Instruction	Defined in MIPS ISA
BC1FL	Branch on FP False Likely	MIPS32
BC1TL	Branch on FP True Likely	MIPS32

5.9.6 Miscellaneous Instructions

The MIPS ISA defines various miscellaneous instructions that conditionally move one CPU general register to another, based on an FPU condition code.Table 5-23 lists these conditional move instructions.

Table 5-23 CPU Conditional Move on FPU True/False Instructions

Mnemonic	Instruction	Defined in MIPS ISA
MOVN	Move Conditional on FP False	MIPS32
MOVZ	Move Conditional on FP True	MIPS32

5.10 Valid Operands for FPU Instructions

The floating point unit arithmetic, conversion, and operand move instructions operate on formatted values with different precision and range limits and produce formatted values for results. Each representable value in each format has a binary encoding that is read from or stored to memory. The *fmt* field of the instruction encodes the operand format required for the instruction. A conversion instruction specifies the result type in the *function* field; the result of other operations is given in the same format as the operands. The encodings of the *fmt fmt3* field are shown in Table 5-24.

Table 5-24 FPU Operand Format Field (*fmt*) Encoding

fmt	Instruction Mnemonic	Size		Data Type
		Name	Bits	
0-15	Reserved			
16	S	single	32	Floating point
17	D	double	64	Floating point
18-19	Reserved			
20	W	word	32	Fixed point
21	Reserved			
22–31	Reserved			

The result of an instruction using operand formats marked **U** in Table 5-24 is not currently specified by this architecture and causes an exception. They are being held for future extensions to the architecture. The exact exception mechanism used is processor specific. Most implementations report this as an Unimplemented Operation for a Floating Point exception, although some implementations report these combinations as Reserved Instruction exceptions.

In Table 5-25, the result of an instruction using operand formats marked **i** are invalid and an attempt to execute such an instruction has an undefined result.

Table 5-25 Valid Formats for FPU Operations

Mnemonic	Operation	Operand Fmt				COP1 Function Value
		Float		Fixed		
		S	D	W	L	
ABS	Absolute value	•	•	U	U	5
ADD	Add	•	•	U	U	0
C.*cond*	Floating Point compare	•	•	U	U	48–63
CEIL.W	Convert to word fixed point, round toward +∞	•	•	i	i	14
CVT.D	Convert to double floating point	•	i	•	•	33
CVT.S	Convert to single floating point	i	•	•	•	32
CVT.W	Convert to 32-bit fixed point	•	•	i	i	36
DIV	Divide	•	•	U	U	3
FLOOR.W	Convert to word fixed point, round toward -∞	•	•	i	i	15
MOV	Move Register	•	•	i	i	6
MOVC	FP Move conditional on condition	•	•	i	i	17
MOVN	FP Move conditional on GPR≠zero	•	•	i	i	19
MOVZ	FP Move conditional on GPR=zero	•	•	i	i	18
MSUB	Multiply-Subtract	•	•	U	U	
MUL	Multiply	•	•	U	U	2
NEG	Negate	•	•	U	U	7
ROUND.W	Convert to word fixed point, round to nearest/even	•	•	i	i	12
SQRT	Square Root	•	•	U	U	4
SUB	Subtract	•	•	U	U	1
TRUNC.W	Convert to word fixed point, round toward zero	•	•	i	i	13
Key: • – Valid. **U** – Unimplemented or Reserved. **i** – Invalid.						

5.11 FPU Instruction Formats

An FPU instruction is a single 32-bit aligned word. FP instruction formats are shown in Figures 5-10 through 5-23.

In these figures, variables are labelled in lowercase, such as *offset*. Constants are labelled in uppercase, as are numerals. Following these figures, Table 5-26 explains the fields used in the instruction layouts. Note that the same field may have different names in different instruction layouts.

The field name is mnemonic to the function of that field in the instruction layout. The opcode tables and the instruction encode discussion use the canonical field names: *opcode*, *fmt*, *nd*, *tf*, and *function*. The remaining fields are not used for instruction encode.

5.11.1 Implementation Note

When present, the destination FPR specifier may be in the *fs*, *ft* or *fd* field.

Figure 5-10 I-Type (Immediate) FPU Instruction Format

31 26	25 21	20 16	15 0
opcode	base	ft	offset
6	5	5	16

Immediate: Load/Store using register + offset addressing

Figure 5-11 R-Type (Register) FPU Instruction Format

31 26	25 21	20 16	15 11	10 6	5 0
COP1	fmt	ft	fs	fd	function
6	5	5	5	5	6

Register: Two-register and Three-register formatted arithmetic operations

Figure 5-12 Register-Immediate FPU Instruction Format

31 26	25 21	20 16	15 11	0
COP1	sub	rt	fs	0
6	5	5	5	11

Register Immediate: Data transfer, CPU ↔ FPU register

Figure 5-13 Condition Code, Immediate FPU Instruction Format

31 26	25 21	20 18	17	16	15 0
COP1	BCC1	cc	nd	tf	offset
6	5	3	1	1	16

Condition Code, Immediate: Conditional branches on FPU cc using PC + offset

Figure 5-14 Formatted FPU Compare Instruction Format

31 26	25 21	20 16	15 11	10 8	7 6	5 0
COP1	fmt	ft	fs	cc	0	function
6	5	5	5	3	2	6

Register to Condition Code: Formatted FP compare

Figure 5-15 FP RegisterMove, Conditional Instruction Format

31 26	25 21	20 18	17	16	15 11	10 6	5 0
COP1	fmt	cc	0	tf	fs	fd	MOVCF
6	5	3	1	1	5	5	6

Condition Code, Register FP: FPU register move-conditional on FP, cc

Figure 5-16 Condition Code, Register Integer FPU Instruction Format

31 26	25 21	20 18	17	16	15 11	10 6	5 0
SPECIAL	rs	cc	0	tf	rd	0	MOVCI
6	5	3	1	1	5	5	6

Condition Code, Register Integer: CPU register move-conditional on FP, cc

Table 5-26 FPU Instruction Format Fields

Field	Description
BC1	Branch Conditional instruction subcode (op=COP1).
base	CPU register: base address for address calculations.
COP1	Coprocessor 1 primary *opcode* value in *op* field.
cc	*Condition Code* specifier; for architectural levels prior to MIPS IV, this must be set to zero.
fd	FPU register: destination (arithmetic, loads, move-to) or source (stores, move-from).
fmt	Destination and/or operand type (*format*) specifier.
fr	FPU register: source.
fs	FPU register: source.
ft	FPU register: source (for stores, arithmetic) or destination (for loads).
function	Field specifying a function within a particular *op* operation code.
index	CPU register that holds the index address component for address calculations.
MOVC	Value in *function* field for a conditional move. There is one value for the instruction when *op*=COP1, another value for the instruction when *op*=SPECIAL.
nd	Nullify delay. If set, the branch is Likely, and the delay slot instruction is not executed.
offset	Signed *offset* field used in address calculations.
op	Primary operation code (see COP1, COP1X, LWC1, SWC1, LDC1, SDC1, SPECIAL).
rd	CPU register: destination.
rs	CPU register: source.
rt	CPU register: can be either source or destination.
SPECIAL	*SPECIAL* primary *opcode* value in *op* field.
sub	Operation subcode field for COP1 register immediate-mode instructions.
tf	True/False. The condition from an FP compare that is tested for equality with the *tf* bit.

Instruction Bit Encodings

A.1 Instruction Encodings and Instruction Classes

Instruction encodings are presented in this section; field names are printed here and throughout the book in *italics*.

When encoding an instruction, the primary *opcode* field is encoded first. Most *opcode* values completely specify an instruction that has an *immediate* value or offset.

Opcode values that do not specify an instruction instead specify an instruction class. Instructions within a class are further specified by values in other fields. For instance, *opcode* REGIMM specifies the *immediate* instruction class, which includes conditional branch and trap *immediate* instructions.

A.2 Instruction Bit Encoding Tables

This section provides various bit encoding tables for the instructions of the MIPS32 ISA.

Figure A-1 shows a sample encoding table and the instruction *opcode* field this table encodes. Bits 31..29 of the *opcode* field are listed in the leftmost columns of the table. Bits 28..26 of the *opcode* field are listed along the topmost rows of the table. Both decimal and binary values are given, with the first three bits designating the row, and the last three bits designating the column.

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

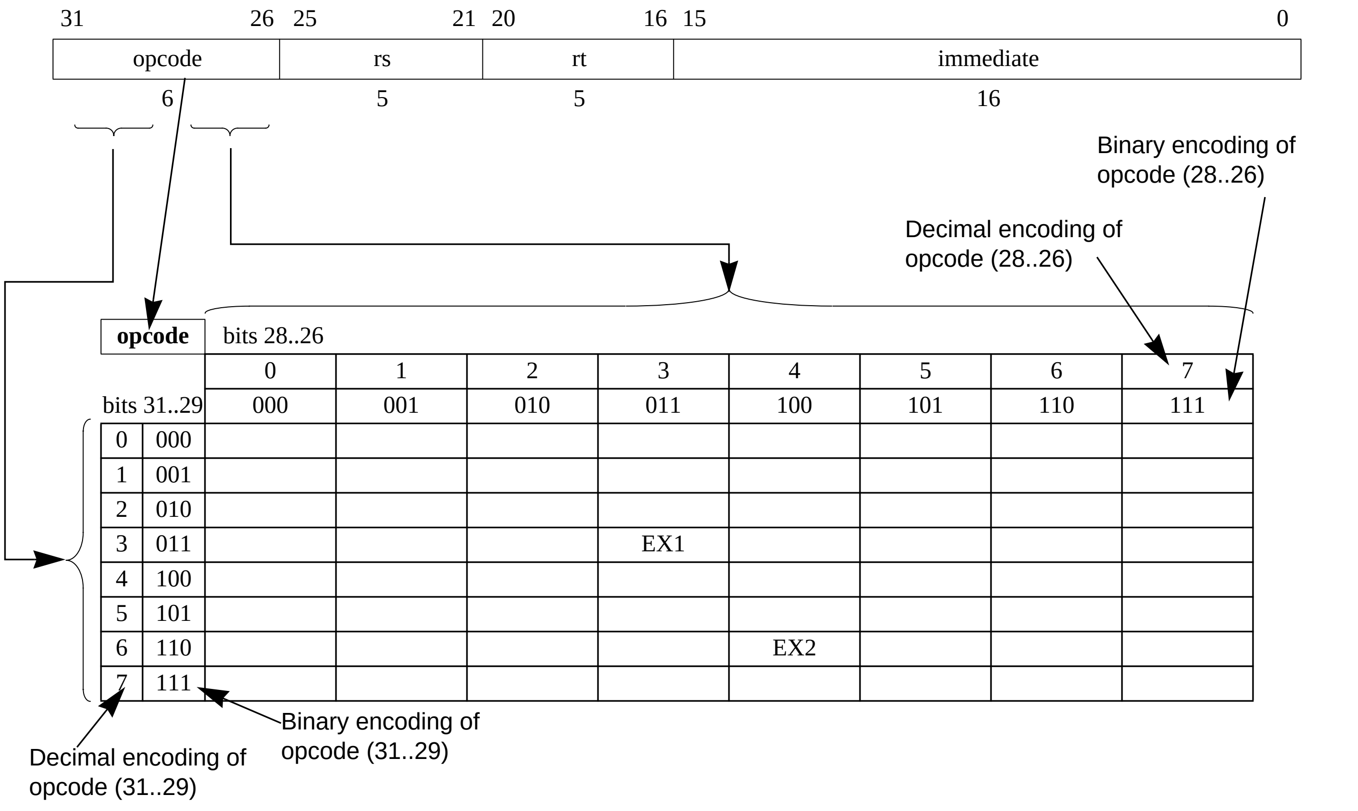


Figure A-1 Sample Bit Encoding Table

Tables A-2 through A-15 describe the encoding used for the MIPS32 ISA. Table A-1 describes the meaning of the symbols used in the tables.

Table A-1 Symbols Used in the Instruction Encoding Tables

Symbol	Meaning
*	Operation or field codes marked with this symbol are reserved for future use. Executing such an instruction must cause a Reserved Instruction Exception.
δ	(Also *italic* field name.) Operation or field codes marked with this symbol denotes a field class. The instruction word must be further decoded by examining additional tables that show values for another instruction field.
β	Operation or field codes marked with this symbol represent a valid encoding for a higher-order MIPS ISA level. Executing such an instruction must cause a Reserved Instruction Exception.
θ	Operation or field codes marked with this symbol are available to licensed MIPS partners. To avoid multiple conflicting instruction definitions, the partner must notify MIPS Technologies, Inc. when one of these encodings is used. If no instruction is encoded with this value, executing such an instruction must cause a Reserved Instruction Exception (*SPECIAL2* 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).
σ	Field codes marked with this symbol represent an EJTAG support instruction and implementation of this encoding is optional for each implementation. If the encoding is not implemented, executing such an instruction must cause a Reserved Instruction Exception. If the encoding is implemented, it must match the instruction encoding as shown in the table.
ε	Operation or field codes marked with this symbol are reserved for MIPS Application Specific Extensions. If the ASE is not implemented, executing such an instruction must cause a Reserved Instruction Exception.

Table A-1 Symbols Used in the Instruction Encoding Tables

Symbol	Meaning
ϕ	Operation or field codes marked with this symbol are obsolete and will be removed from a future revision of the MIPS32 ISA. Software should avoid using these operation or field codes.

Table A-2 MIPS32 Encoding of the Opcode Field

opcode	*bits 28..26*								
		0	1	2	3	4	5	6	7
bits 31..29		000	001	010	011	100	101	110	111
0	000	*SPECIAL* δ	*REGIMM* δ	J	JAL	BEQ	BNE	BLEZ	BGTZ
1	001	ADDI	ADDIU	SLTI	SLTIU	ANDI	ORI	XORI	LUI
2	010	*COP0* δ	*COP1* δ	*COP2* θδ	*COP3* θδ	BEQL ϕ	BNEL ϕ	BLEZL ϕ	BGTZL ϕ
3	011	β	β	β	β	*SPECIAL2* δ	JALX ε	ε	*
4	100	LB	LH	LWL	LW	LBU	LHU	LWR	β
5	101	SB	SH	SWL	SW	β	β	SWR	CACHE
6	110	LL	LWC1	LWC2 θ	PREF	β	LDC1	LDC2 θ	β
7	111	SC	SWC1	SWC2 θ	*	β	SDC1	SDC2 θ	β

Table A-3 MIPS32 *SPECIAL* Opcode Encoding of Function Field

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	SLL	*MOVCI* δ	SRL	SRA	SLLV	*	SRLV	SRAV
1	001	JR	JALR	MOVZ	MOVN	SYSCALL	BREAK	*	SYNC
2	010	MFHI	MTHI	MFLO	MTLO	β	*	β	β
3	011	MULT	MULTU	DIV	DIVU	β	β	β	β
4	100	ADD	ADDU	SUB	SUBU	AND	OR	XOR	NOR
5	101	*	*	SLT	SLTU	β	β	β	β
6	110	TGE	TGEU	TLT	TLTU	TEQ	*	TNE	*
7	111	β	*	β	β	β	*	β	β

Table A-4 MIPS32 *REGIMM* Encoding of rt Field

rt	*bits 18..16*								
		0	1	2	3	4	5	6	7
bits 20..19		000	001	010	011	100	101	110	111
0	00	BLTZ	BGEZ	BLTZL ϕ	BGEZL ϕ	*	*	*	*
1	01	TGEI	TGEIU	TLTI	TLTIU	TEQI	*	TNEI	*
2	10	BLTZAL	BGEZAL	BLTZALL ϕ	BGEZALL ϕ	*	*	*	*
3	11	*	*	*	*	*	*	*	*

Table A-5 MIPS32 *SPECIAL2* Encoding of Function Field

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	MADD	MADDU	MUL	θ	MSUB	MSUBU	θ	θ
1	001	θ	θ	θ	θ	θ	θ	θ	θ
2	010	θ	θ	θ	θ	θ	θ	θ	θ
3	011	θ	θ	θ	θ	θ	θ	θ	θ
4	100	CLZ	CLO	θ	θ	β	β	θ	θ
5	101	θ	θ	θ	θ	θ	θ	θ	θ
6	110	θ	θ	θ	θ	θ	θ	θ	θ
7	111	θ	θ	θ	θ	θ	θ	θ	SDBBP σ

Table A-6 MIPS32 *MOVCI* Encoding of tf Bit

tf	*bit 16*	
	0	1
	MOVF	MOVT

Table A-7 MIPS32 *COPz* 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	MFCz	β	CFCz	*	MTCz	β	CTCz	*
1	01	*BCz* δ	*	*	*	*	*	*	*
2	10	*CO* δ							
3	11								

Table A-8 MIPS32 COPz Encoding of rt Field When rs=*BCz*

rt	*bits 16*	
bit 17	0	1
0	BCzF	BCzT
1	BCzFL ϕ	BCzTL ϕ

Table A-9 MIPS32 *COP0* 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	MFC0	β	*	*	MTC0	β	*	*
1	01	*	*	*	*	*	*	*	*
2	10	*CO* δ							
3	11								

Table A-10 MIPS32 *COP0* Encoding of Function Field When rs=*CO*

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	*	TLBR	TLBWI	*	*	*	TLBWR	*
1	001	TLBP	*	*	*	*	*	*	*
2	010	*	*	*	*	*	*	*	*
3	011	ERET	*	*	*	*	*	*	DERET σ
4	100	WAIT	*	*	*	*	*	*	*
5	101	*	*	*	*	*	*	*	*
6	110	*	*	*	*	*	*	*	*
7	111	*	*	*	*	*	*	*	*

Table A-11 MIPS32 *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	MFC1	β	CFC1	*	MTC1	β	CTC1	*
1	01	*BC1* δ	ε	ε⊥	*	*	*	*	*
2	10	*S* δ	*D* δ	*	*	*W* δ	β	β	*
3	11	*	*	*	*	*	*	*	*

Table A-12 MIPS32 *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	ADD	SUB	MUL	DIV	SQRT	ABS	MOV	NEG
1	001	β	β	β	β	ROUND.W	TRUNC.W	CEIL.W	FLOOR.W
2	010	*	*MOVCF* δ	MOVZ	MOVN	*	β	β	*
3	011	*	*	*	*	ε	ε	ε	ε
4	100	*	CVT.D	*	*	CVT.W	β	β	*
5	101	*	*	*	*	*	*	*	*
6	110	C.F	C.UN	C.EQ	C.UEQ	C.OLT	C.ULT	C.OLE	C.ULE
7	111	C.SF	C.NGLE	C.SEQ	C.NGL	C.LT	C.NGE	C.LE	C.NGT

Table A-13 MIPS32 *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	ADD	SUB	MUL	DIV	SQRT	ABS	MOV	NEG
1	001	β	β	β	β	ROUND.W	TRUNC.W	CEIL.W	FLOOR.W
2	010	*	*MOVCF* δ	MOVZ	MOVN	*	β	β	*
3	011	*	*	*	*	ε	ε	ε	ε
4	100	CVT.S	*	*	*	CVT.W	β	*	*
5	101	*	*	*	*	*	*	*	*
6	110	C.F	C.UN	C.EQ	C.UEQ	C.OLT	C.ULT	C.OLE	C.ULE
7	111	C.SF	C.NGLE	C.SEQ	C.NGL	C.LT	C.NGE	C.LE	C.NGT

Table A-14 MIPS32 *COP1* Encoding of Function Field When rs=*W*

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.S	CVT.D	*	*	*	*	ε	*
5	101	*	*	*	*	*	*	*	*
6	110	*	*	*	*	*	*	*	*
7	111	*	*	*	*	*	*	*	*

Table A-15 MIPS32 *COP1* Encoding of tf Bit When rs=*S, D, or PS*, Function=*MOVCF*

tf	*bit 16*	
	0	1
	MOVF.fmt	MOVT.fmt

Revision History

Revision	Date	Description
0.95	March 12, 2001	External review copy of reorganized and updated architecture documentation.