MIPS® Architecture For Programmers Volume I-A: Introduction to the MIPS64® Architecture

Document Number: MD00083
Revision 3.02
March 21, 2011

MIPS Technologies, Inc.
955 East Arques Avenue
Sunnyvale, CA 94085-4521

Template: nB1.03, Built with tags: 2B ARCH FPU_PS FPU_PSandARCH MIPS64

Contents

Figures

Tables

Chapter 1

About This Book

The MIPS® Architecture For Programmers Volume I-A: Introduction to the MIPS64® Architecture comes as part of a multi-volume set.

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

1.1 Typographical Conventions

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

1.1.1 Italic Text

- is used for *emphasis*

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

- is used for the memory access types, such as *cached* and *uncached*

1.1.2 Bold Text

- represents a term that is being **defined**

- is used for **bits** and **fields** that are important from a hardware perspective (for instance, **register** bits, which are not programmable but accessible only to hardware)

- is used for ranges of numbers; the range is indicated by an ellipsis. For instance, **5..1** indicates numbers 5 through 1

- is used to emphasize **UNPREDICTABLE** and **UNDEFINED** behavior, as defined below.

1.1.3 Courier Text

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

1.2 UNPREDICTABLE and UNDEFINED

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

1.2.1 UNPREDICTABLE

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

UNPREDICTABLE results or operations have several implementation restrictions:

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

- **UNPREDICTABLE** operations must not read, write, or modify the contents of memory or internal state which is inaccessible in the current processor mode. For example, **UNPREDICTABLE** operations executed in user mode must not access memory or internal state that is only accessible in Kernel Mode or Debug Mode or in another process

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

1.2.2 UNDEFINED

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

UNDEFINED operations or behavior has one implementation restriction:

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

1.2.3 UNSTABLE

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

UNSTABLE values have one implementation restriction:

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

1.3 Special Symbols in Pseudocode Notation

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

Table 1.1 Symbols Used in Instruction Operation Statements

Symbol	Meaning
$\leftarrow$	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.
0bn	A constant value *n* in base *2*. For instance 0b100 represents the binary value 100 (decimal 4).
0xn	A constant value *n* in base *16*. For instance 0x100 represents the hexadecimal value 100 (decimal 256).
$x_{y..z}$	Selection of bits *y* through *z* of bit string *x*. Little-endian bit notation (rightmost bit is 0) is used. If *y* is less than *z*, this expression is an empty (zero length) bit string.
$+, -$	2’s complement or floating point arithmetic: addition, subtraction
$*, \times$	2’s complement or floating point multiplication (both used for either)
div	2’s complement integer division

Table 1.1 Symbols Used in Instruction Operation Statements (Continued)

Symbol	Meaning
mod	2’s complement modulo
/	Floating point division
<	2’s complement less-than comparison
>	2’s complement greater-than comparison
≤	2’s complement less-than or equal comparison
≥	2’s complement greater-than or equal comparison
nor	Bitwise logical NOR
xor	Bitwise logical XOR
and	Bitwise logical AND
or	Bitwise logical OR
GPRLEN	The length in bits (32 or 64) of the CPU general-purpose registers
GPR[x]	CPU general-purpose register *x*. The content of *GPR[0]* is always zero. In Release 2 of the Architecture, GPR[x] is a short-hand notation for *SGPR[SRSCtl*$_{CSS}$*, x]*.
SGPR[s,x]	In Release 2 of the Architecture and subsequent releases, multiple copies of the CPU general-purpose registers may be implemented. *SGPR[s,x]* refers to GPR set *s*, register *x*.
FPR[x]	Floating Point operand register *x*
FCC[CC]	Floating Point condition code CC. *FCC[0]* has the same value as *COC[1]*.
FPR[x]	Floating Point (Coprocessor unit 1), general register *x*
CPR[z,x,s]	Coprocessor unit *z*, general register *x*, select *s*
CP2CPR[x]	Coprocessor unit 2, general register *x*
CCR[z,x]	Coprocessor unit *z*, control register *x*
CP2CCR[x]	Coprocessor unit 2, control register *x*
COC[z]	Coprocessor unit *z* condition signal
Xlat[x]	Translation of the MIPS16e GPR number *x* into the corresponding 32-bit GPR number
BigEndianMem	Endian mode as configured at chip reset (0 →Little-Endian, 1 → Big-Endian). Specifies the endianness of the memory interface (see LoadMemory and StoreMemory pseudocode function descriptions), and the endianness of Kernel and Supervisor mode execution.
BigEndianCPU	The endianness for load and store instructions (0 → Little-Endian, 1 → Big-Endian). In User mode, this endianness may be switched by setting the *RE* bit in the *Status* register. Thus, BigEndianCPU may be computed as (BigEndianMem XOR ReverseEndian).
ReverseEndian	Signal to reverse the endianness of load and store instructions. This feature is available in User mode only, and is implemented by setting the *RE* bit of the *Status* register. Thus, ReverseEndian may be computed as (SR_{RE} and User mode).
LLbit	Bit of **virtual** state used to specify operation for instructions that provide atomic read-modify-write. *LLbit* is set when a linked load occurs and is tested by the conditional store. It is cleared, during other CPU operation, when a store to the location would no longer be atomic. In particular, it is cleared by exception return instructions.

Table 1.1 Symbols Used in Instruction Operation Statements (Continued)

<table>
<tr><th>Symbol</th><th>Meaning</th></tr>
<tr><td><b>I</b>:,
<b>I+n</b>:,
<b>I-n</b>:</td><td>This occurs as a prefix to <i>Operation</i> 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 <b>I</b>. 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 <b>I</b>, 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 <b>I+1</b>.
The effect of pseudocode statements for the current instruction labelled <b>I+1</b> appears to occur “at the same time” as the effect of pseudocode statements labeled <b>I</b> for the following instruction. Within one pseudocode sequence, the effects of the statements take place in order. However, between sequences of statements for different instructions that occur “at the same time,” there is no defined order. Programs must not depend on a particular order of evaluation between such sections.</td></tr>
<tr><td>PC</td><td>The <i>Program Counter</i> 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 <i>PC</i> during an instruction time. If no value is assigned to <i>PC</i> during an instruction time by any pseudocode statement, it is automatically incremented by either 2 (in the case of a 16-bit MIPS16e instruction) or 4 before the next instruction time. A taken branch assigns the target address to the <i>PC</i> during the instruction time of the instruction in the branch delay slot.
In the MIPS Architecture, the PC value is only visible indirectly, such as when the processor stores the restart address into a GPR on a jump-and-link or branch-and-link instruction, or into a Coprocessor 0 register on an exception. The PC value contains a full 64-bit address all of which are significant during a memory reference.</td></tr>
<tr><td>ISA Mode</td><td>In processors that implement the MIPS16e Application Specific Extension or the microMIPS base architectures, the <i>ISA Mode</i> is a single-bit register that determines in which mode the processor is executing, as follows:
<table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>The processor is executing 32-bit MIPS instructions</td></tr><tr><td>1</td><td>The processor is executing MIIPS16e instructions</td></tr></table>
In the MIPS Architecture, the ISA Mode value is only visible indirectly, such as when the processor stores a combined value of the upper bits of PC and the ISA Mode into a GPR on a jump-and-link or branch-and-link instruction, or into a Coprocessor 0 register on an exception.</td></tr>
<tr><td>PABITS</td><td>The number of physical address bits implemented is represented by the symbol PABITS. As such, if 36 physical address bits were implemented, the size of the physical address space would be $2^{PABITS} = 2^{36}$ bytes.</td></tr>
<tr><td>SEGBITS</td><td>The number of virtual address bits implemented in a segment of the address space is represented by the symbol SEGBITS. As such, if 40 virtual address bits are implemented in a segment, the size of the segment is $2^{SEGBITS} = 2^{40}$ bytes.</td></tr>
<tr><td>FP32RegistersMode</td><td>Indicates whether the FPU has 32-bit or 64-bit floating point registers (FPRs). In MIPS32 Release 1, the FPU has 32 32-bit FPRs in which 64-bit data types are stored in even-odd pairs of FPRs. In MIPS64, (and optionally in MIPS32 Release2 and MIPSr3) the FPU has 32 64-bit FPRs in which 64-bit data types are stored in any FPR.

In MIPS32 Release 1 implementations, <b>FP32RegistersMode</b> 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 <b>FP32RegisterMode</b> is computed from the FR bit in the <i>Status</i> 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 <b>FP32RegistersMode</b> is computed from the FR bit in the <i>Status</i> register.</td></tr>
</table>

Table 1.1 Symbols Used in Instruction Operation Statements (Continued)

Symbol	Meaning
InstructionInBranchDelaySlot	Indicates whether the instruction at the Program Counter address was executed in the delay slot of a branch or jump. This condition reflects the *dynamic* state of the instruction, not the *static* state. That is, the value is false if a branch or jump occurs to an instruction whose PC immediately follows a branch or jump, but which is not executed in the delay slot of a branch or jump.
SignalException(exception, argument)	Causes an exception to be signaled, using the exception parameter as the type of exception and the argument parameter as an exception-specific argument). Control does not return from this pseudocode function—the exception is signaled at the point of the call.

1.4 For More Information

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

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

Chapter 2

The MIPS Architecture: An Introduction

2.1 MIPS Instruction Set 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®, MIPS64® and microMIPS™ 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.

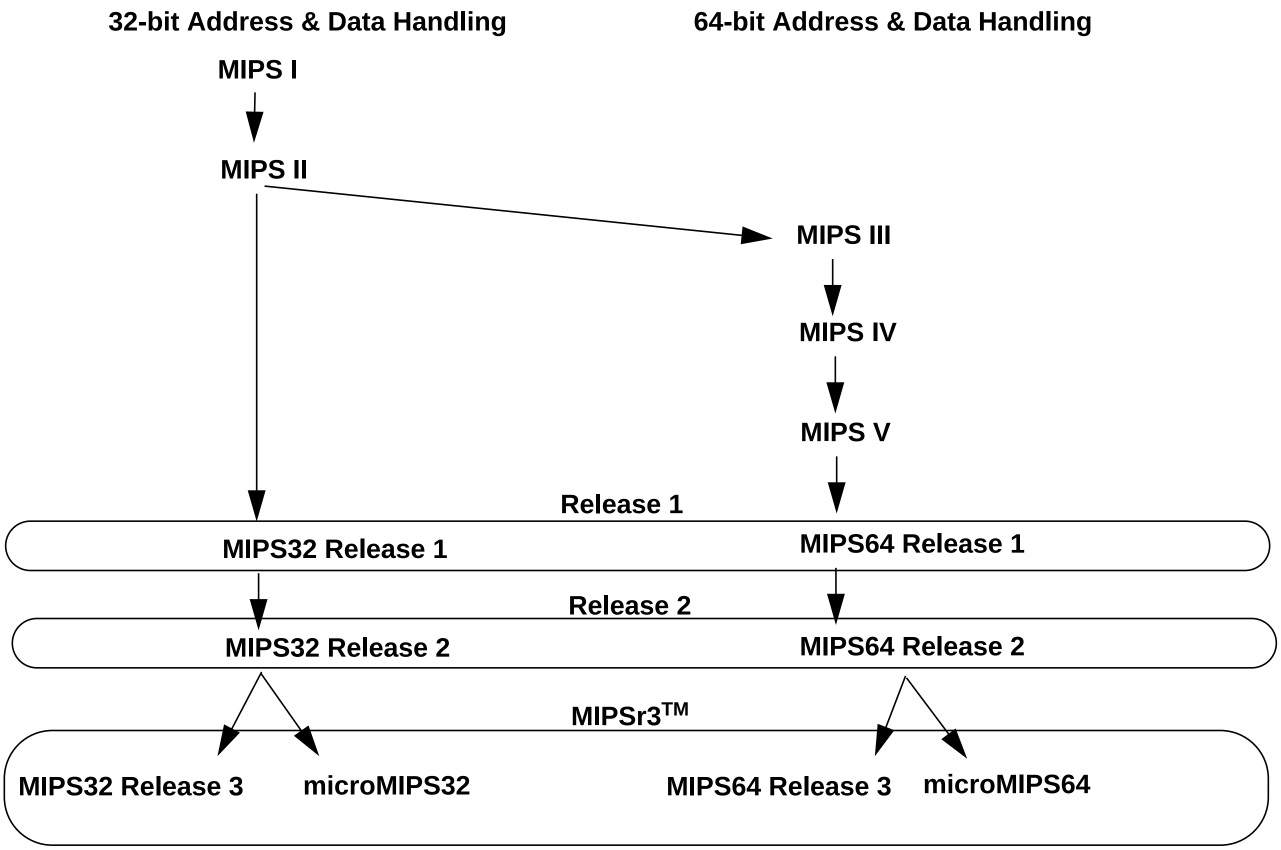
The MIPS32 and MIPS64 Architectures are intended to address the need for a high-performance but cost-sensitive MIPS instruction set. The MIPS32 Architecture is based on the MIPS II ISA, adding selected instructions from MIPS III, MIPS IV, and MIPS V to improve the efficiency of generated code and of 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.

The microMIPS32 and microMIPS64 Architectures deliver the same functionality of MIPS32 and MIPS64 with the additional benefit of smaller codesizes. The microMIPS architectures are supersets of MIPS32/MIPS64 architectures, with almost the same sets of 32-bit sized instructions and additional 16-bit instructions to help with codesize. microMIPS is especially compelling for systems in which the cost of memory dominate the entire bill of materials cost.

Unlike the earlier versions of the architectures, microMIPS supplies assembler-source code compatibility with its predecessors instead of binary compatibility.

Figure 2-1 MIPS Architectures



2.1.2 Architectural Evolution

The evolution of an architecture is a dynamic process that takes into account both the need to provide a stable platform for implementations, as well as new market and application areas that demand new capabilities. Enhancements to an architecture are appropriate when they:

- are applicable to a wide market
- provide long-term benefit
- maintain architectural scalability
- are standardized to prevent fragmentation
- are a superset of the existing architecture

The MIPS Architecture community constantly evaluates suggestions for architectural changes and enhancements against these criteria. New releases of the architecture, while infrequent, are made at appropriate points, following these criteria. At present, there are three releases of the MIPS Architecture: Release 1 (the original version of the MIPS64 Architecture) ; Release 2 which was added in 2002 and Release 3 (called MIPSr3^{TM}) which was added in 2010.

2.1.2.1 Release 2 of the MIPS64 Architecture

Enhancements included in Release 2 of the MIPS64 Architecture are:

- Vectored interrupts: This enhancement provides the ability to vector interrupts directly to a handler for that interrupt. Vectored interrupts are an option in Release 2 implementations and the presence of that option is denoted by the Config3$_{VInt}$ bit.

- Support for an external interrupt controller: This enhancement reconfigures the on-core interrupt logic to take full advantage of an external interrupt controller. This support is an option in Release 2 implementations and the presence of that option is denoted by the Config3$_{EIC}$ bit.

- Programmable exception vector base: This enhancement allows the base address of the exception vectors to be moved for exceptions that occur when Status$_{BEV}$ is 0. Doing so allows multi-processor systems to have separate exception vectors for each processor, and allows any system to place the exception vectors in memory that is appropriate to the system environment. This enhancement is required in a Release 2 implementation.

- Atomic interrupt enable/disable: Two instructions have been added to atomically enable or disable interrupts, and return the previous value of the *Status* register. These instructions are required in a Release 2 implementation.

- The ability to disable the *Count* register for highly power-sensitive applications. This enhancement is required in a Release 2 implementation.

- GPR shadow registers: This addition provides the addition of GPR shadow registers and the ability to bind these registers to a vectored interrupt or exception. Shadow registers are an option in Release 2 implementations and the presence of that option is denoted by a non-zero value in SRSCtl$_{HSS}$. While shadow registers are most useful when either vectored interrupts or support for an external interrupt controller is also implemented, neither is required.

- Field, Rotate and Shuffle instructions: These instructions add additional capability in processing bit fields in registers. These instructions are required in a Release 2 implementation.

- Explicit hazard management: This enhancement provides a set of instructions to explicitly manage hazards, in place of the cycle-based SSNOP method of dealing with hazards. These instructions are required in a Release 2 implementation.

- Access to a new class of hardware registers and state from an unprivileged mode. This enhancement is required in a Release 2 implementation.

- Coprocessor 0 Register changes: These changes add or modify CP0 registers to indicate the existence of new and optional state, provide L2 and L3 cache identification, add trigger bits to the Watch registers, and add support for 64-bit performance counter count registers. This enhancement is required in a Release 2 implementation.

- Support for 64-bit coprocessors with 32-bit CPUs: These changes allow a 64-bit coprocessor (including an FPU) to be attached to a 32-bit CPU. This enhancement is optional in a Release 2 implementation.

- New Support for Virtual and Physical Memory: These changes provide support for a 1KByte page size, and the ability to support physical addresses larger than 36 bits. Both changes are optional in Release 2 implementations, and support is denoted by Config3$_{SP}$ (for 1KB page support) and Config3$_{LPA}$ (for larger physical address support).

2.1.2.2 Releases 2.5+ of the MIPS64 Architecture

Some optional features were added after Revision 2.5:

- TLB pages larger than 256MB are supported. This feature allows large regions to be mapped with fewer TLB entries, especially within devices with very large memory systems.

- Support for a MMU with more than 64 TLB entries. This feature aids in reducing the frequency of TLB misses.

- Scratch registers within Coprocessor0 for kernel mode software. This feature aids in quicker exception handling by not requiring the saving of usermode registers onto the stack before kernelmode software uses those registers.

- A MMU configuration which supports both larger set-associative TLBs and variable page-sizes. This feature aids in reducing the frequency of TLB misses.

- The CDMM memory scheme for the placement of small I/O devices into the physical address space. This scheme allows for efficient placement of such I/O devices into a small memory region.

- An EIC interrupt mode where the EIC controller supplies a 16-bit interrupt vector. This allows different interrupts to share code.

- The PAUSE instruction to deallocate a (virtual) processor when arbitration for a lock doesn’t succeed. This allows for lower power consumption as well as lower snoop traffic when multiple (virtual) processors are arbitrating for a lock.

- More flavors of memory barriers that are available through stype field of the SYNC instruction. The newer memory barriers attempt to minimize the amount of pipeline stalls while doing memory synchronization operations.

2.1.2.3 MIPSr3^{TM} Architecture

MIPSr3™ is a family of architectures which includes Release 3.0 of the MIPS64 Architecture as well as the first release of the microMIPS64 architecture.

Enhancements included in MIPSr3™ Architecture are:

- The microMIPS^{TM} instruction set.

    - This instruction set contains both 16-bit and 32-bit sized instructions.

    - This mixed size ISA has all of the functionality of MIPS64 while also delivering smaller code sizes.

    - microMIPS is assembler source code compatible with MIPS64.

    - microMIPS replaces the MIPS16e^{TM} ASE.

    - microMIPS is an additional base instruction set architecture that is supported along with MIPS64.

    - A device can implement either base ISA or both. The ISA field of *Config3* denotes which ISA is implemented.
    - A device can implement any other ASE with either base architecture.[1]
    - microMIPS shares the same privileged resource architecture with MIPS64.
    - Branch Likely instructions are not supported in the microMIPS hardware architecture. Instead the microMIPS toolchain replaces these instructions with equivalent code sequences.
- A more flexible version of the Context Register that can point to any power-of-two sized data structure. This optional feature is denoted by CTXTC field of *Config3.*
- Additional protection bits in the TLB entries that allow for non-executable and write-only virtual pages. This optional feature is denoted by RXI field of *Config3.*

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

In addition to the MIPS 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 MIPS64 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 MIPS64 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 Compliance and Subsetting

To be compliant with the MIPS64 Architecture, designs must implement a set of required features, as described in this document set. To allow flexibility in implementations, the MIPS64 Architecture does provide subsetting rules. An implementation that follows these rules is compliant with the MIPS64 Architecture as long as it adheres strictly to the rules, and fully implements the remaining instructions.Supersetting of the MIPS64 Architecture is only allowed by adding functions to the *SPECIAL2* major opcode, by adding control for co-processors via the *COP2*, *LWC2*, *SWC2*, *LDC2*, and/or *SDC2*, or via the addition of approved Application Specific Extensions.

[1] Except for MIPS16e.

Note: The use of COP3 as a customizable coprocessor has been removed in the Release 2 of the MIPS64 architecture. The use of the COP3 is now reserved for the future extension of the architecture.

The instruction set subsetting rules are as follows:

- All CPU instructions must be implemented - no subsetting is allowed.

- The FPU and related support instructions, including the MOVF and MOVT CPU instructions, may be omitted. Software may determine if an FPU is implemented by checking the state of the FP bit in the *Config1* CP0 register. If the FPU is implemented, the paired single (PS) format is optional. Software may determine which FPU data types are implemented by checking the appropriate bit in the *FIR* CP1 register. The following allowable FPU subsets are compliant with the MIPS64 architecture:

  - No FPU

  - FPU with S, D, W, and L formats and all supporting instructions

  - FPU with S, D, PS, W, and L formats and all supporting instructions

- Coprocessor 2 is optional and may be omitted. Software may determine if Coprocessor 2 is implemented by checking the state of the C2 bit in the *Config1* CP0 register. If Coprocessor 2 is implemented, the Coprocessor 2 interface instructions (BC2, CFC2, COP2, CTC2, DMFC2, DMTC2, LDC2, LWC2, MFC2, MTC2, SDC2, and SWC2) may be omitted on an instruction-by-instruction basis.

- Implementation of the full 64-bit address space is optional. The processor may implement 64-bit data and operations with a 32-bit only address space. In this case, the MMU acts as if 64-bit addressing is always disabled. Software may determine if the processor implements a 32-bit or 64-bit address space by checking the AT field in the *Config* CP0 register.

- Supervisor Mode is optional. If Supervisor Mode is not implemented, bit 3 of the *Status* register must be ignored on write and read as zero.

- The standard TLB-based memory management unit may be replaced with:

  - a simpler MMU (e.g., a Fixed Mapping MMU or a Block Address Translation MMU or a Base-Bounds MMU).

  - The Dual TLB MMU - (e.g. FTLB and VTLB MMU described in the *Alternative MMU Organizations* Appendix of Volume III)

  If this is done, the rest of the interface to the Privileged Resource Architecture must be preserved. Software may determine the type of the MMU by checking the MT field in the *Config* CP0 register.

- The Privileged Resource Architecture includes several implementation options and may be subsetted in accordance with those options. An incomplete list of these options include:

  - Interrupt Modes

  - Shadow Register Sets

  - Common Device Memory Map

  - Parity/ECC support

  - UserLocal register
  - ContextConfig register
  - PageGrain register
  - Config1-4 registers
  - Performance Counter, WatchPoint and Trace Registers
  - Cache control/diagnostic registers
  - Kernelmode scratch registers

- Instruction, CP0 Register, and CP1 Control Register fields that are marked “Reserved” or shown as “0” in the description of that field are reserved for future use by the architecture and are not available to implementations. Implementations may only use those fields that are explicitly reserved for implementation dependent use.

- Supported ASEs are optional and may be subsetted out. If most cases, software may determine if a supported ASE is implemented by checking the appropriate bit in the *Config1* or *Config3* CP0 register. If they are implemented, they must implement the entire ISA applicable to the component, or implement subsets that are approved by the ASE specifications.

- EJTAG is optional and may be subsetted out. If it is implemented, it must implement only those subsets that are approved by the EJTAG specification.

- If any instruction is subsetted out based on the rules above, an attempt to execute that instruction must cause the appropriate exception (typically Reserved Instruction or Coprocessor Unusable).

- In MIPSr3 (also called Release 3), there are two architecture branches (MIPS32/64 and microMIPS32/64). A single device is allowed to implement both architecture branches. The Privileged Resource Architecture (COP0) registers do not mode-switch in width (32-bit vs. 64-bit). For this reason, if a device implements both architecture branches, the address/data widths must be consistent. If a device implements MIPS64 and also implements microMIPS, it must implement microMIPS64 not just microMIPS32. Simiarly, If a device implements microMIPS64 and also implements MIPS32/64, it must implement MIPS64 not just MIPS32.

- If both of the architecture branches are implemented (MIPS32/64 and microMIPS32/64) or if MIPS16e is implemented then the JALX instructions are required. If only one branch of the architecture family and MIPS16e is not implemented then the JALX instruction is not implemented. That is, the JALX instruction is required if and only if when ISA mode-switching is possible.

2.3 Components of the MIPS Architecture

2.3.1 MIPS Instruction Set Architecture (ISA)

The MIPS32 and MIPS64 Instruction Set Architectures define a compatible family of instructions dealing with 32-bit data and 64-bit data (respectively) within the framework of the overall MIPS 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.

2.3.2 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.

2.3.3 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.

2.3.4 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.

2.4 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 MIPS64 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 MIPS64 architecture is decoupled from specific hardware implementations, leaving microprocessor designers free to create their own hardware designs within the framework of the architectural definition.

2.5 Relationship between the MIPSr3 Architectures

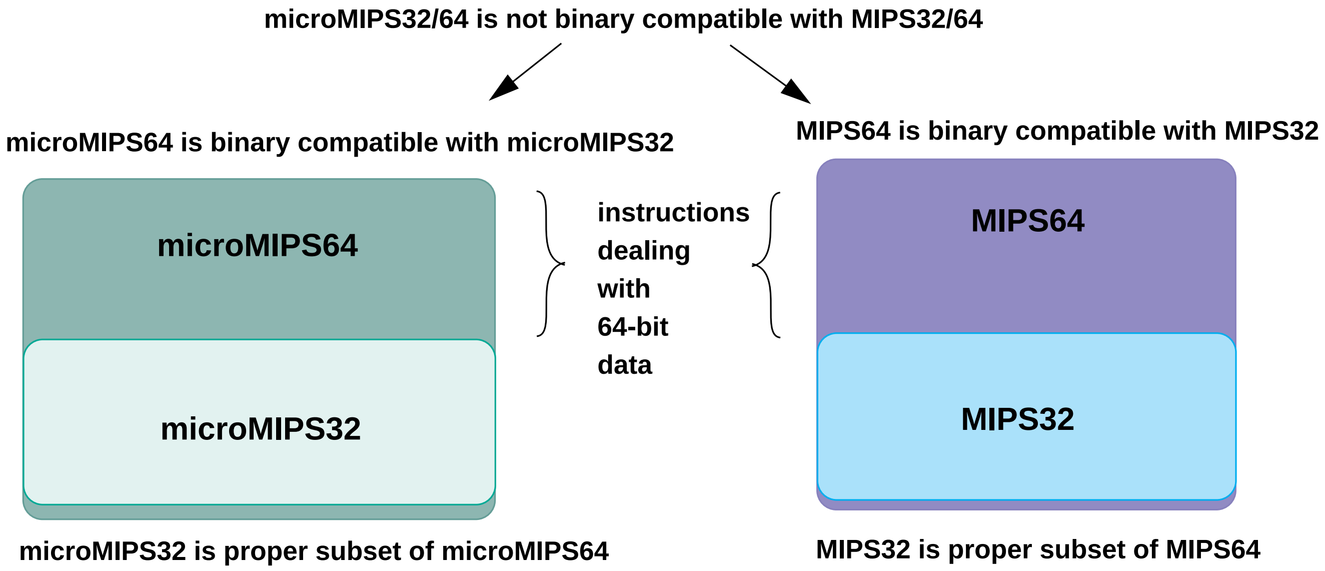
The MIPS Architectures evolved as a compromise between software and hardware resources. The MIPS has a family of related architectures. Within each “branch of the family”, the architecture guarantees object-code compatibility for User-Mode programs executed on any MIPS processor.

MIPS32 and MIPS64 form one branch of the architecture family. 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.

Similarly, microMIPS32 and microMIPS64 form another branch of the architecture family. In User Mode microMIPS64 processors are backward-compatible with their microMIPS predecessors. As such, the microMIPS Architecture is a strict subset of the MIPS64 Architecture.

The relationship between the binary representations of the architectures is shown in Figure 2-2.

Figure 2-2 Relationship of the Binary Representations of MIPSr3 Architectures


microMIPS32/64 is not binary compatible with MIPS32/64
microMIPS64 is binary compatible with microMIPS32
MIPS64 is binary compatible with MIPS32
microMIPS64
microMIPS32
instructions dealing with 64-bit data
MIPS64
MIPS32
microMIPS32 is proper subset of microMIPS64
MIPS32 is proper subset of MIPS64

As of 2010, there are two branches of the architecture family - the MIPS32/64 branch and the microMIPS32/64 branch. For these two branches, some levels of compatibility are available:

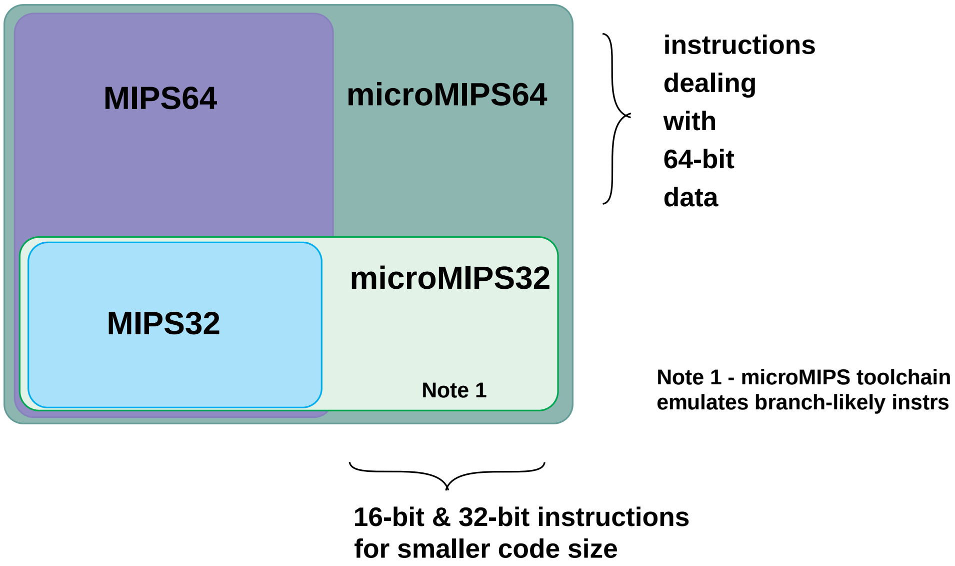
1. The microMIPS32/64 branch supplies a superset of the functionality that is available from the MIPS32/64 branch. The additional functionality that the microMIPS branch delivers is smaller code size.

2. It is allowed for implementations to implement both branches of the architecture family for compatibility reasons. For such implementations, the architectures define methods of switching from one instruction set to the other. This allows one binary program to use both instruction sets or call a library that is using the other instruction set.

3. At the assembler source code level, the two architecture branches are fully compatible. That is, all of the MIPS32/64 assembler instruction mnemonics and directives are fully usable and understood by the microMIPS32/64 toolchains.

The relationships between the assembler source-code representations of the architectures is shown in Figure 2-3.

Figure 2-3 Relationships of the Assembler Source Code Representations of the MIPSr3 Architectures


MIPS64
microMIPS64
instructions dealing with 64-bit data
MIPS32
microMIPS32
Note 1
Note 1 - microMIPS toolchain emulates branch-likely instrs
16-bit & 32-bit instructions for smaller code size

2.6 Pipeline Architecture

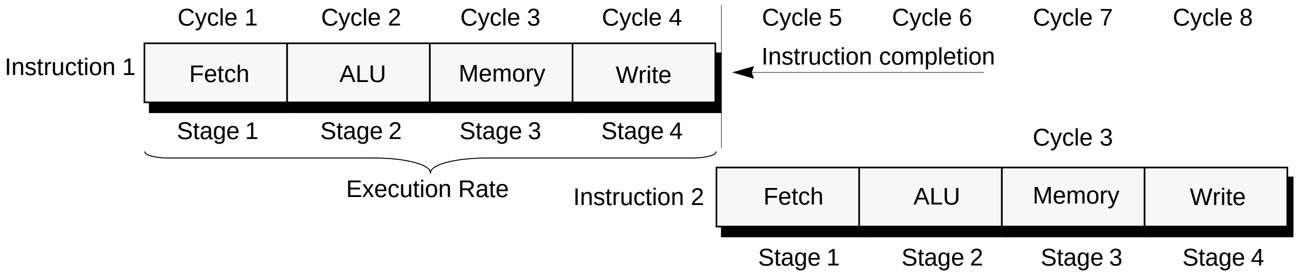
This section describes the basic pipeline architecture, along with two types of improvements: superpipelines and superscalar pipelines. (Pipelining and multiple issuing are not defined by the ISA, but are implementation dependent.)

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 2-4:

- Fetch
- Arithmetic operation
- Memory access
- Write back

Figure 2-4 One-Deep Single-Completion Instruction Pipeline



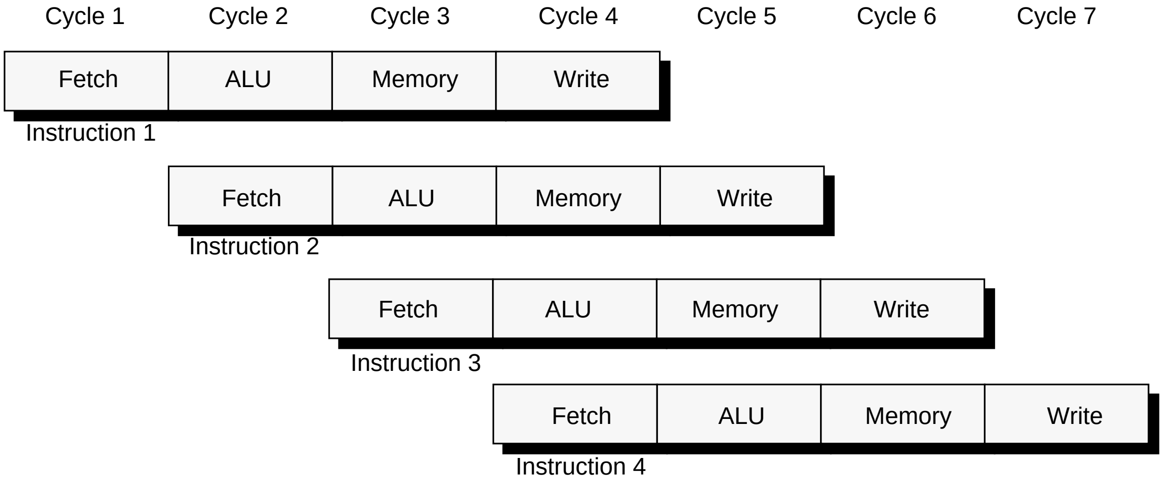
In the example shown in Figure 2-4, each stage takes one processor clock cycle to complete. Thus it takes four clock cycles (ignoring delays or stalls) for the instruction to complete. In this example, the **execution rate** of the pipeline is one instruction every four clock cycles. Conversely, because only a single execution can be fetched before completion, only one stage is active at any time.

2.6.2 Parallel Pipeline

Figure 2-5 illustrates a remedy for the **latency** (the time it takes to execute an instruction) inherent in the pipeline shown in Figure 2-4.

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 2-5 are executed at a rate four times that of the pipeline shown in Figure 2-4.

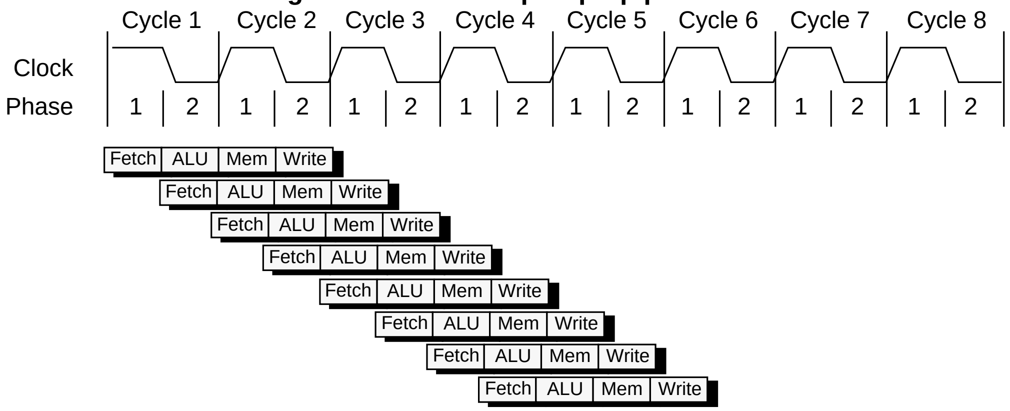
Figure 2-5 Four-Deep Single-Completion Pipeline



2.6.3 Superpipeline

Figure 2-6 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.

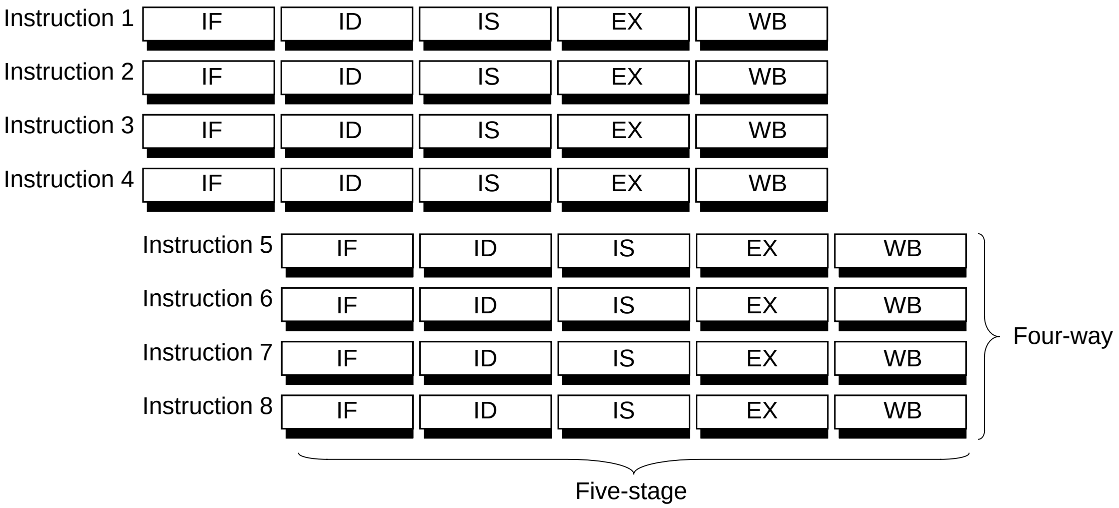
Figure 2-6 Four-Deep Superpipeline



2.6.4 Superscalar Pipeline

A **superscalar** architecture also allows more than one instruction to be completed each clock cycle. Figure 2-7 shows a four-way, five-stage superscalar pipeline.

Figure 2-7 Four-Way Superscalar Pipeline



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

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*)[2]

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*)[2]
- 64-bit double-precision floating point (.fmt type *D*)
- 32-bit Word fixed point (.fmt type *W*)

[2] The CPU Doubleword and FPU floating point paired-single and Long fixed point data formats are available in a Release 1 implementation of the MIPS64 Architecture, or in a Release 2 (or subsequent releases) implementation that includes a 64-bit floating point unit

- 64-bit Long fixed point (.fmt type *L*)[2]

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 a Release 1 implementation of the MIPS64 Architecture, and on all Release 2 (and subsequent releases) implementations of the 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 MIPS64 Architecture defines the following CPU registers:

- 32 64-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.

A MIPS64 processor always produces a 64-bit result, even for those instructions which are architecturally defined to operate on 32 bits. Such instructions typically sign-extend their 32-bit result into 64 bits. In so doing, 32-bit programs work as expected, even though the registers are actually 64 bits wide rather than 32.

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.

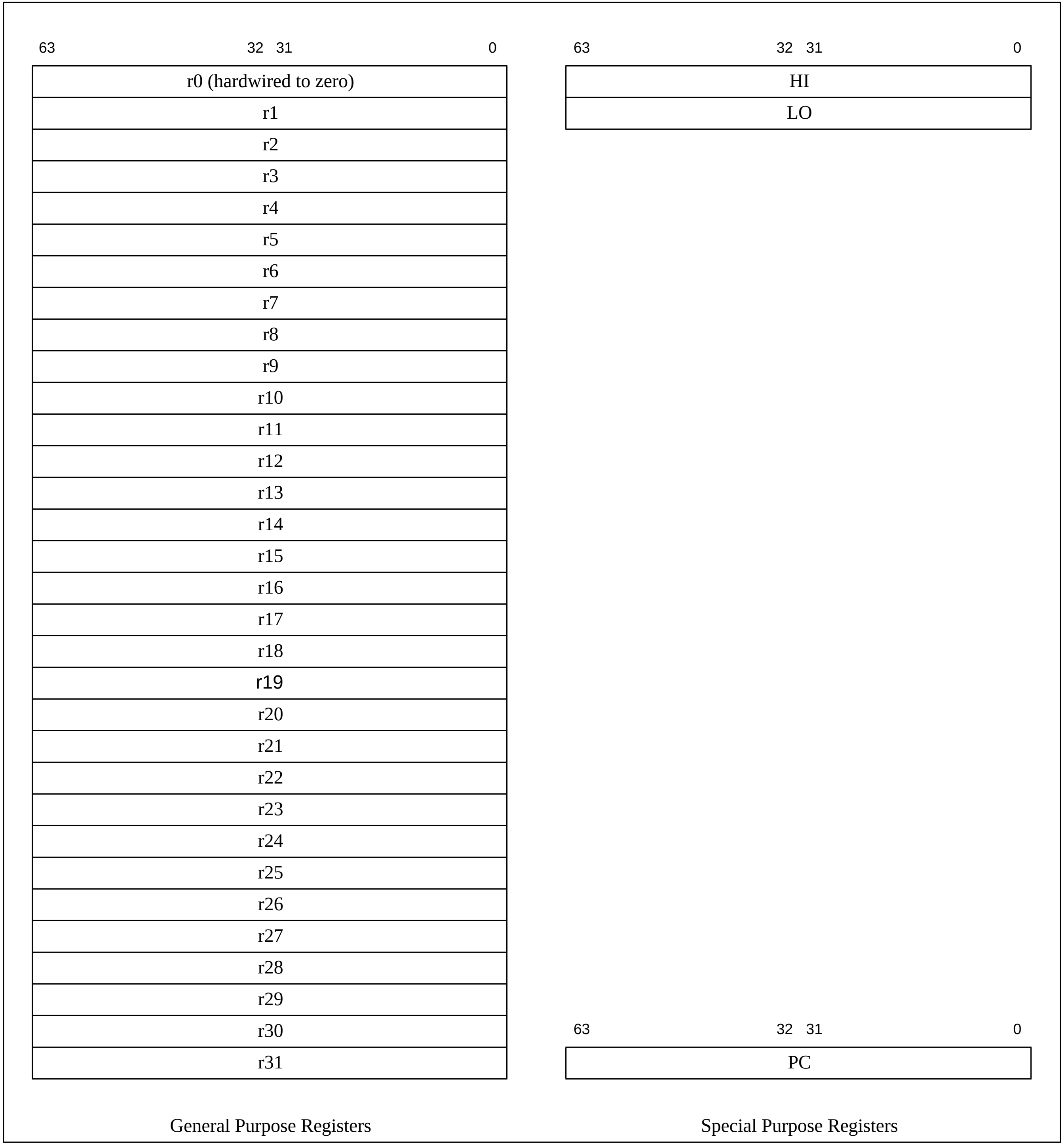
2.8.4.2 CPU Special-Purpose Registers

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 2-8 shows the layout of the CPU registers.

Figure 2-8 CPU Registers



2.8.5 FPU Registers

The MIPS64 Architecture defines the following FPU registers:

- 32 floating point registers (FPRs). These registers are 32 bits wide in a 32-bit FPU and 64 bits wide on a 64-bit FPU.

- Five FPU control registers are used to identify and control the FPU.

- Eight floating point condition codes that are part of the *FCSR* register

In Release 1 of the Architecture, 64-bit floating point units were supported only by implementations of the MIPS64 Architecture. Similarly, implementations of MIPS32 of the Architecture only supported 32-bit floating point units. In Release 2 of the Architecture and subsequent releases, a 64-bit floating point unit is optional on implementations of both the MIPS32 and MIPS64 Architectures.

A 32-bit floating point unit contains 32 32-bit FPRs, each of which is capable of storing a 32-bit data type. Double-precision (type D) data types are stored in even-odd pairs of FPRs, and the long-integer (type L) and paired single (type PS) data types are not supported. Figure 2-9 shows the layout of these registers.

A 64-bit floating point unit contains 32 64-bit FPRs, each of which is capable of storing any data type. For compatibility with 32-bit FPUs, the FR bit in the CP0 *Status* register is used by a MIPS64 Release 1, or any Release 2 (or subsequent releases) processor that supports a 64-bit FPU to configure the FPU in a mode in which the FPRs are treated as 32 32-bit registers, each of which is capable of storing only 32-bit data types. In this mode, the double-precision floating point (type D) data type is stored in even-odd pairs of FPRs, and the long-integer (type L) and paired single (type PS) data types are not supported.

Figure 2-10 shows the layout of the FPU Registers when the FR bit in the CP0 Status register is 1; Figure 2-11 shows the layout of the FPU Registers when the FR bit in the CP0 Status register is 0.

Figure 2-9 FPU Registers for a 32-bit FPU

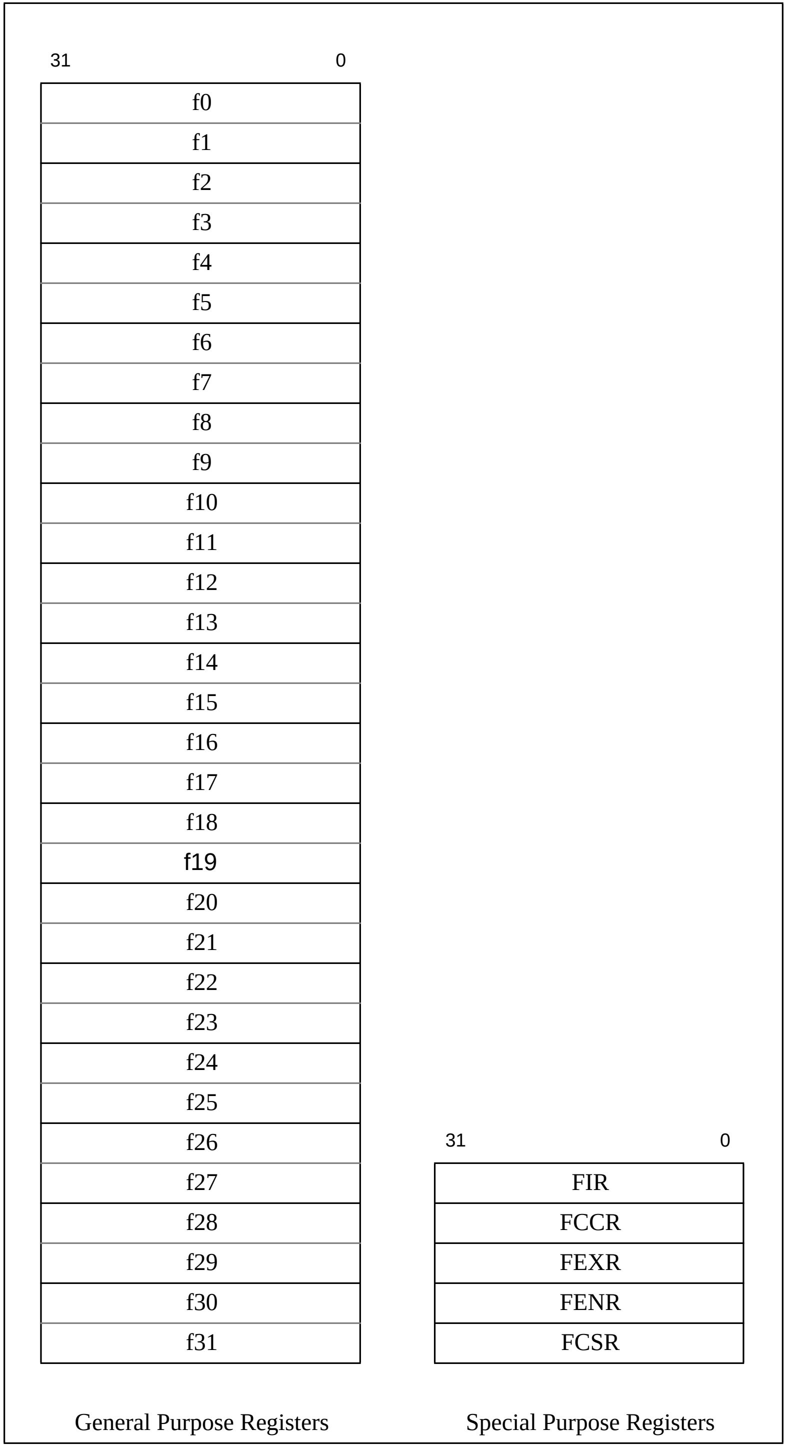


Figure 2-10 FPU Registers for a 64-bit FPU if $Status_{FR}$ is 1

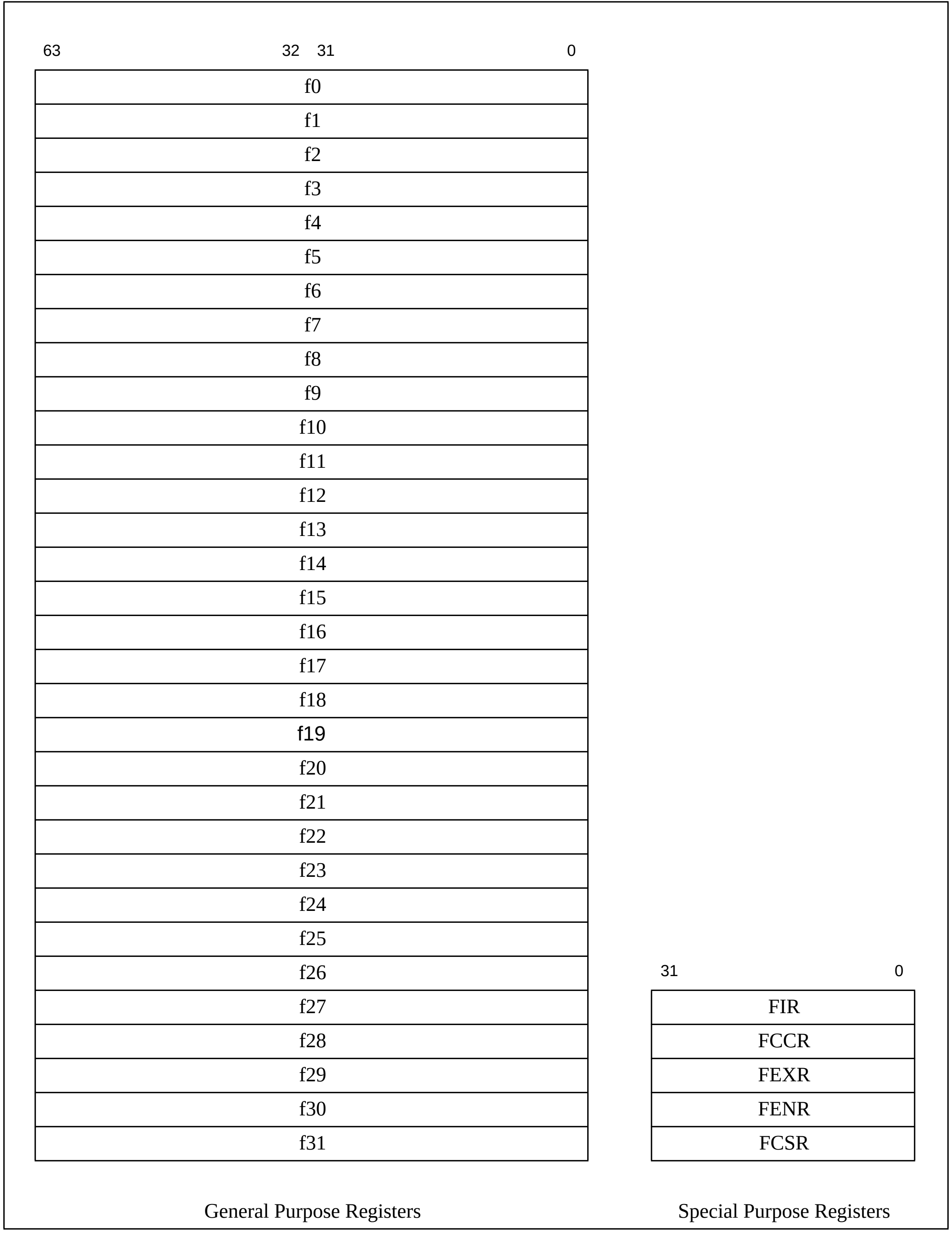
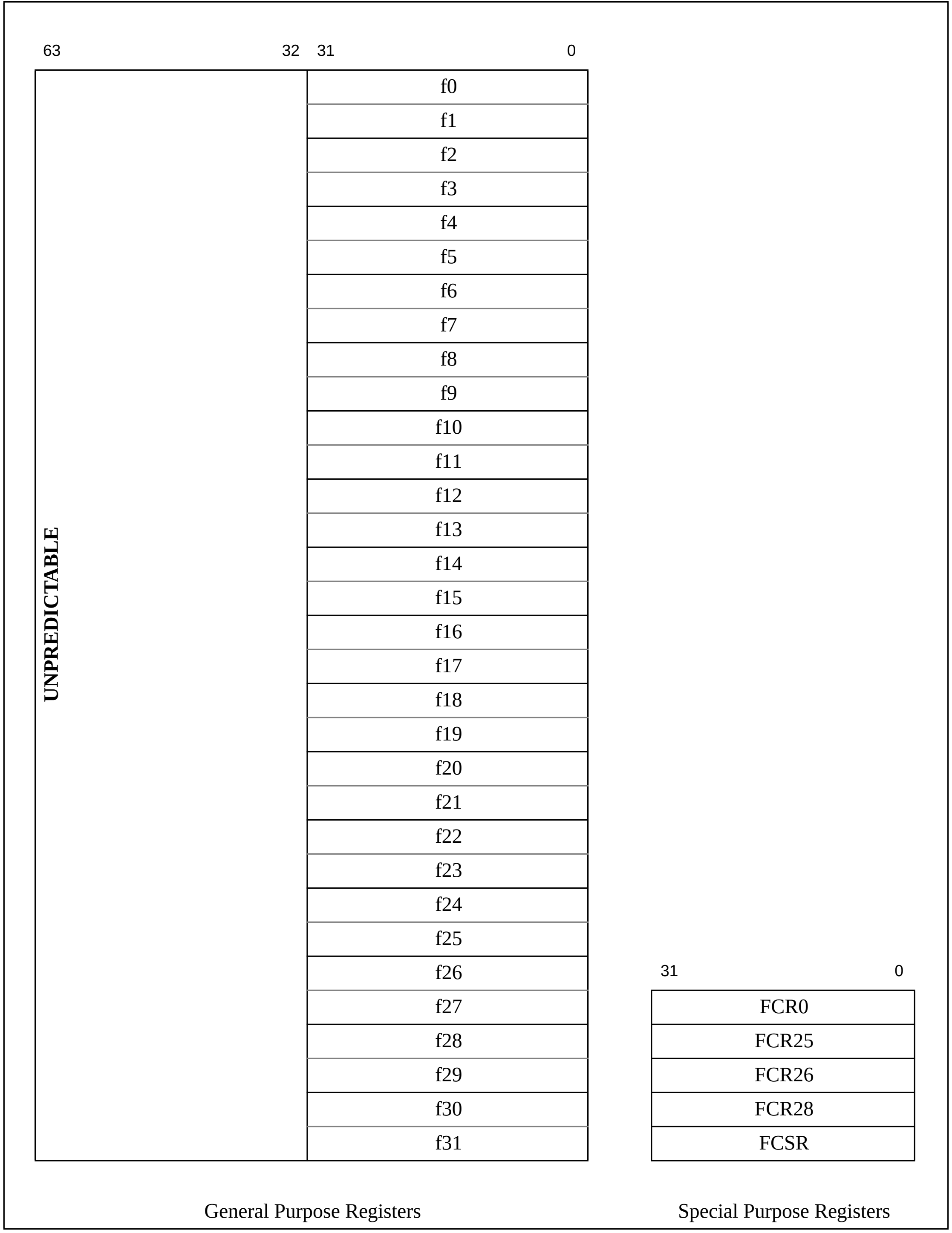


Figure 2-11 FPU Registers for a 64-bit FPU if Status_{FR} is 0



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:

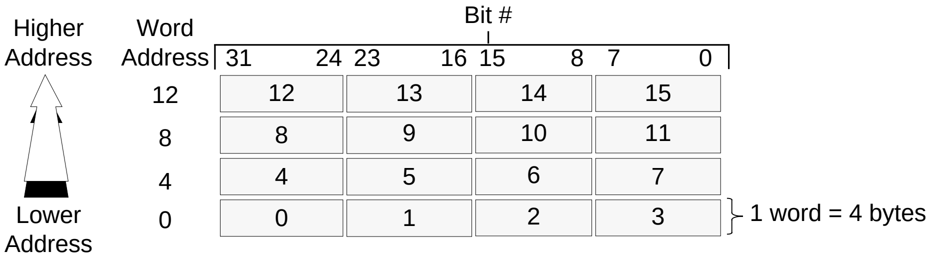
- Big-Endian Order
- Little-Endian Order
- MIPS Bit Endianness

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 2-12 and 2-13 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 2-12 shows this configuration.

Figure 2-12 Big-Endian Byte Ordering

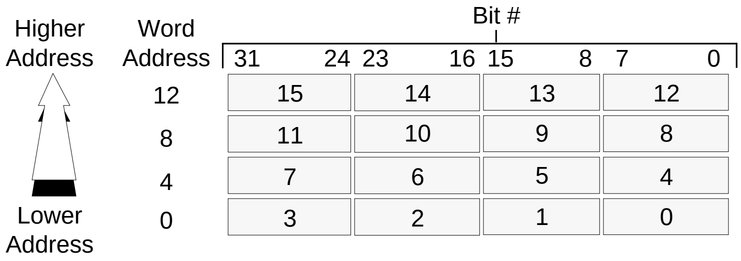


Word Address	31 24	23 16	15 8	7 0
12	12	13	14	15
8	8	9	10	11
4	4	5	6	7
0	0	1	2	3

2.8.6.2 Little-Endian Order

When configured in **little-endian order**, byte 0 is always the least-significant (right-hand) byte. Figure 2-13 shows this configuration.

Figure 2-13 Little-Endian Byte Ordering



Word Address	31 24	23 16	15 8	7 0
12	15	14	13	12
8	11	10	9	8
4	7	6	5	4
0	3	2	1	0

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.

2-14 shows big-endian and 2-15 shows little-endian byte ordering in doublewords.

Figure 2-14 Big-Endian Data in Doubleword Format

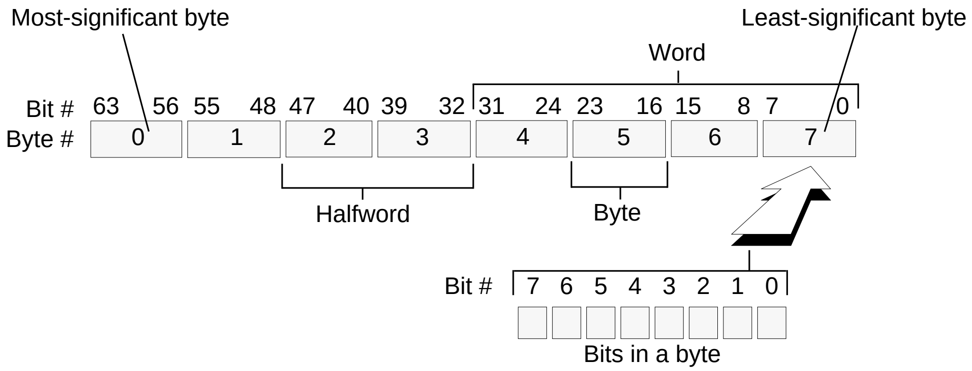
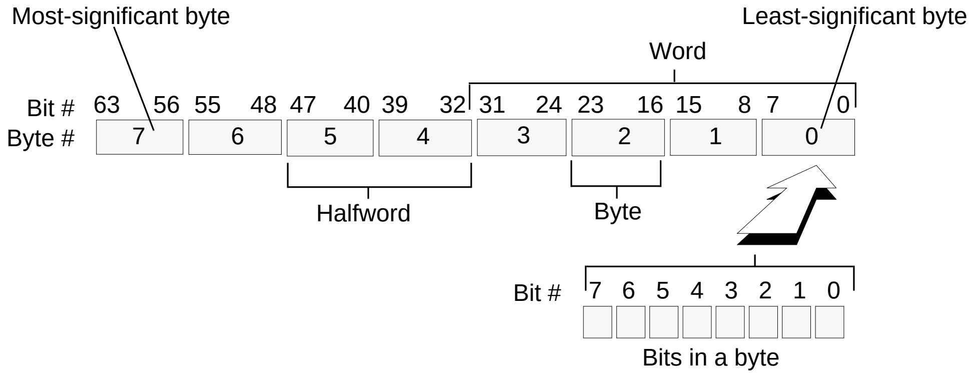


Figure 2-15 Little-Endian Data in Doubleword Format



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 2.1 Unaligned Load and Store Instructions

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

2-16 show a big-endian access of a misaligned word that has byte address 3, and 2-17 shows a little-endian access of a misaligned word that has byte address 1.[3]

Figure 2-16 Big-Endian Misaligned Word Addressing

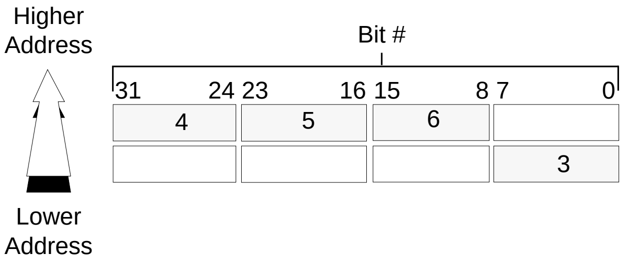
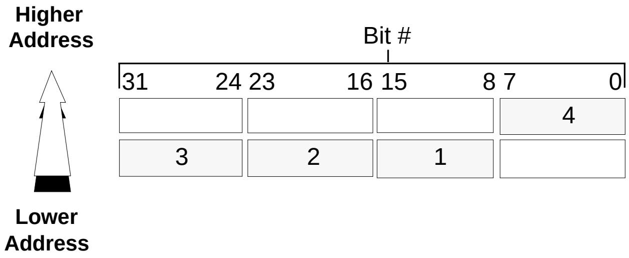


Figure 2-17 Little-Endian Misaligned Word Addressing



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 Cacheability and Coherency Attribute (*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.

[3] 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.

Prefetches for data and instructions are allowed. Speculative prefetching of data that may never be used or instructions which may never be executed are allowed.

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 Cacheability and Coherency Attributes and Access Types

Memory access types are specified by architecturally-defined and implementation-specific Cacheability and Coherency Attribute bits (*CCA*s) kept in TLB entries.

Slightly different cacheability and coherency attributes 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.

2.8.11 Instruction Fetches

2.8.11.1 Instruction fields layout

For MIPS instructions, the layout of the bit fields within the instructions stays the same regardless of the endianness mode in which the processor is executing. The MIPS architecture only uses Little-Endian bit orderings. Bit 0 of an instruction is always the right-most bit within the instruction while bit 31 is always the left-most bit within a 32-bit instruction. The major opcode is always the left-most 6 bits within the instruction.

2.8.11.2 MIPS32 and MIPS64 Instruction placement and endianness

For the MIPS32 and MIPS64 base architectures, instructions are always 32 bits. All instructions are naturally aligned in memory (address bits 1:0 are 0b00).

Instruction words are always placed in memory according to the endianness.

Figure 2-18 shows an example where the width of external memory is 64-bits (two words) and the processor is executing in little-endian mode and the instructions are placed in memory for little-endian execution. In this case, the less significant address is the the right-most word of the dword while the more significant address is the left-most word within the dword.

Figure 2-18 Two instructions placed in a 64-bit wide, little-endian memory

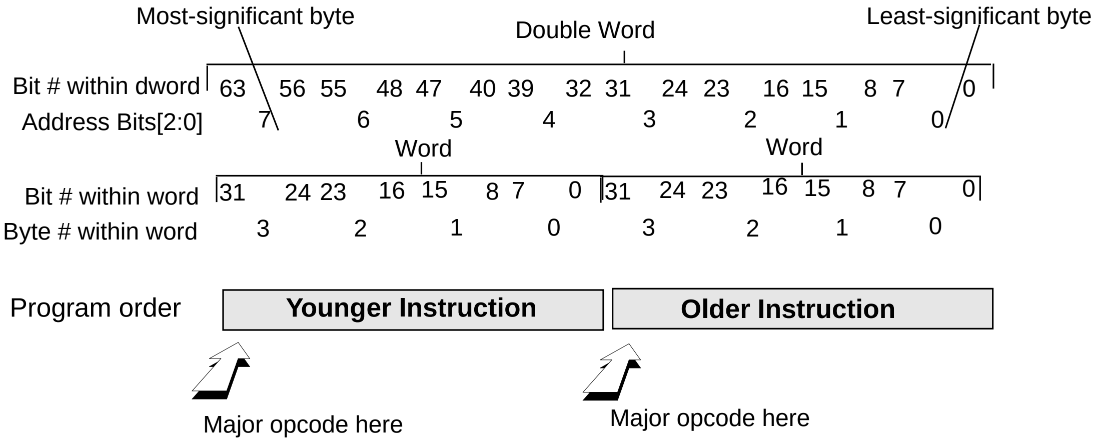
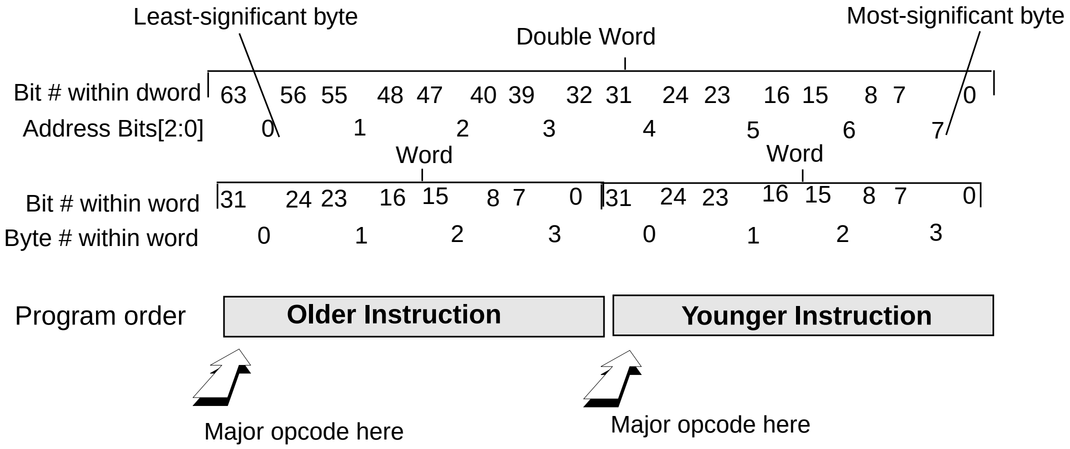


Figure 2-19 shows the equivalent Big-Endian example where the less significant address refers to the left-most word within the dword and the more significant address refers to the right-most word within the dword. In both BE and LE examples, the bit locations within the instruction words has not changed. The location of the major opcode is always at the left-most bits within the word.

Figure 2-19 Two instructions placed in a 64-bit wide, big-endian memory



on. The major opcode is always the left-most 6 bits within the instruction.

2.8.11.3 Instruction fetches using uncached access to memory without side-effects

Memory regions having no access side-effects can be read an infinite amount of times without changing the value received. For such regions accessed with uncached instruction fetches, the following behaviors are allowed:

It is allowed for the fetch transfer size for uncached memory access to be larger than one instruction word. In this case, it is implementation specific whether multiple instruction fetches are done to the same memory location. It is not required for the processor to have a register to buffer the un-used instructions of the transfer for subsequent execution.

Speculative instruction fetches are allowed. Table 2.2 list some types of speculative instruction fetches.

Table 2.2 Speculative instruction fetches

Sequential instructions located after branch/jump fetched before the branch/jump taken/not-taken decision has been determined.
Predicted branch/jump target addresses fetched before branch/jump taken/not-taken decision has been determined or before when target address has been calculated.
Predicted jump target register values before target register has been read.
Predicted return addresses before return register has been read.
Any other type of prefetching ahead of execution.

2.8.11.4 Instruction fetches using uncached access to memory with side-effects

Access side-effects for a memory region might include FIFO behavior, stack behavior or have location-specific behavior (one memory location defining the behavior of another memory location). For such regions accessed with uncached instruction fetches, these are the architectural requirements:

The transfer size can only be one instruction word per instruction fetch.

Speculative instruction fetches are not allowed. The types of instruction fetches listed in Table 2.2 are not allowed.

The architecture defines this memory segment with access side-effects:

- EJTAG Debug Memory space (dmseg). Please refer to MIPS document - *MD00047 EJTAG Specification*.

Beyond this defined segment, the system programmer/designer is reminded that it is possible to memory map an IO device with access side-effects to any uncached memory location, even within segments which the architecture does not define to have access side-effects. For that reason, any implementation which allows behaviors listed in 2.8.11.3 “Instruction fetches using uncached access to memory without side-effects” should restrict software from executing code within any memory region with side-effects.

2.8.11.5 Instruction fetches using cacheable access to memory

The minimum transfer size for cacheable access is one cacheline. The transfer size may be multiple whole cachelines.

Speculative accesses to cacheable memory spaces are allowed as cacheable memory spaces are defined to have no access side-effects. Table 2.2 list some types of speculative instruction fetches.

2.8.11.6 Instruction fetchs and exceptions

Precise exception model for instruction fetches

The MIPS architecture uses the precise exception model for instruction fetches. A precise exception means that for an instruction-sourced exception, the cause of an exception is reported on the exact instruction which the processor has attempted to execute and has caused the exception.

It is not allowed to report an exception for an instruction which could not be executed due to program control flow. For example, if a branch/jump is taken and the instruction after the branch is not to be executed, the address checks (alignment, MMU match/validity, access priviledge) for that not-to-be-executed instruction may not generate any exception.

Instruction fetch exceptions on branch delay-slots

For instructions occupying a branch delay-slot, any exceptions, including those generated by the fetch of that instruction, should report the exception results so that the branch can be correctly replayed upon return from the exception handler.

2.8.11.7 Self-Modified Code

When the processor writes memory with new instructions at run-time, there are some software steps that must be taken to ensure that the new instructions are fetched properly.

1. The path of instruction fetches to external memory may not be the same as the path of data loads/stores to external memory (this feature is known as a Harvard architecture). The new instructions must be flushed out to the first level of the memory hierarchy which is shared by both the instruction fetchs and the data load/stores.

2. The processor must wait until all of the new instructions have actually been written to this shared level of the memory hierarchy.

3. If there are caches which hold instructions between that first shared level of memory hierarchy and the processor pipeline, any stale instructions within those caches must be first invalidated before executing the new instructions.

4. Some processors might implement some type of instruction prefetching. Precautions must be used to ensure that the prefetching does not interfere with the previous steps.

Chapter 3

Application Specific Extensions

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

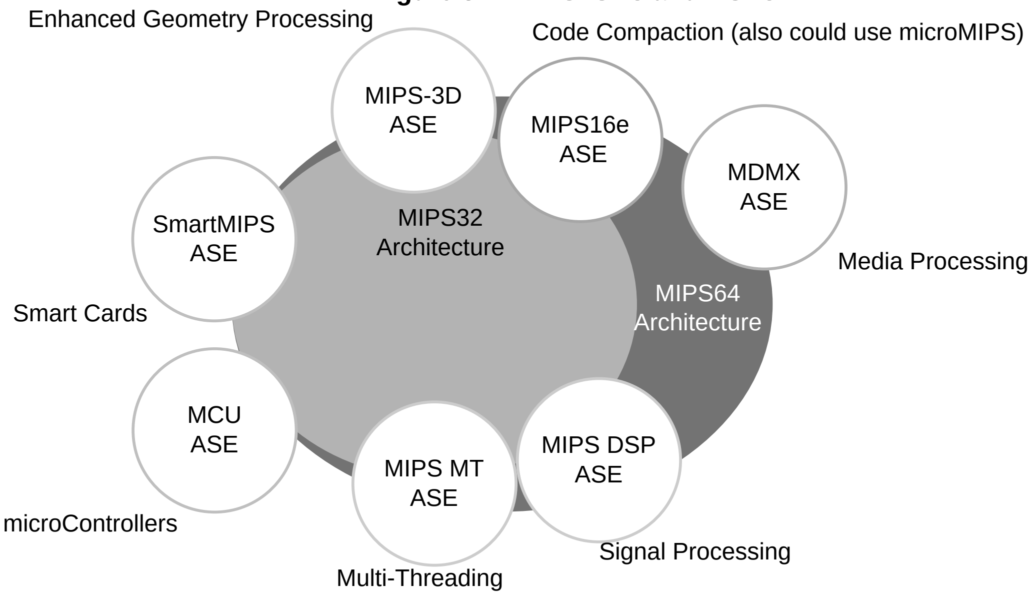
3.1 Description of ASEs

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/MIPS64 and microMIPS32/microMIPS64). 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 3-1shows how ASEs interrelate with the MIPS32 and MIPS64 ISAs.

Figure 3-1 MIPS ISAs and 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	Supported Base Architectures	Use
MIPS16e™	MIPS32 or MIPS64	Code Compaction
MDMX™	MIPS64	Digital Media
MIPS-3D®	MIPS32 or MIPS64	Geometry Processing
SmartMIPS®	MIPS32	Smart Cards and Smart Objects
MIPS® DSP	MIPS32 or MIPS64	Signal Processing
MIPS® MT	MIPS32 or MIPS64	Multi-Threading
MCU	MIPS32 or MIPS64	Fast Interrupt Response & I/O register programming

3.2.1 The MIPS16e™ Application Specific Extension to the MIPS32 and MIPS64 Architecture

The MIPS16e 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 MIPS16e ASE.

The microMIPS Architectures supercedes the MIPS16e ASE as the small code-size solution. microMIPS provides for small code sizes for kernelmode code, floating-point code. These were not available through MIPS16e.

3.2.2 The MDMX™ Application Specific Extension to the MIPS64 Architectures

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. Volume IV-b of this document set describes the MDMX ASE.

3.2.3 The MIPS-3D® Application Specific Extension to the MIPS 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. Volume IV-c of this document set describes the MIPS-3D ASE. Because the MIPS-3D ASE requires a 64-bit floating point unit, it is only available with a Release 1 MIPS64 processor, or a Release 2 (or subsequent releases) processor that includes a 64-bit FPU.

3.2.4 The SmartMIPS® Application Specific Extension to the MIPS32 Architecture

The SmartMIPS ASE extends the MIPS32 Architectures 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. Because the SmartMIPS ASE requires the MIPS32 Architecture, it is not discussed in this document set.

3.2.5 The MIPS® DSP Application Specific Extension to the MIPS Architecture

The MIPS DSP ASE provides enhanced performance of signal-processing applications by providing computational support for fractional data types, SIMD, saturation, and other elements that are commonly used in such applications. Volume IV-e of this document set describes the MIPS DSP ASE.

3.2.6 The MIPS® MT Application Specific Extension to the MIPS Architecture

The MIPS MT ASE provides the architecture to support multi-threaded implementations of the Architecture. This includes support for both virtual processors and lightweight thread contexts. Volume IV-f of this document set describes the MIPS MT ASE.

3.2.7 The MIPS® MCU Application Specific Extension to the MIPS Architecture

The MIPS MCU ASE provides enhanced handling of memory-mapped I/O registers and lower interrupt latencies. Volume IV-g of this document set describes the MIPS MCU 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)
- Selecting the addressing mode (for example, SDXC1, in the FPU)
- Atomic memory update (read-modify-write: for instance, LL/SC)

Regardless of the byte ordering (big- or little-endian), the address of a halfword, word, or doubleword 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 37 for more information on big-endian and little-endian data ordering.

4.1.1.2 Load and Store Access Types

Tables 4.1 and 4.2 list 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

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

Table 4.2 FPU Load and Store Operations Using Register + Register Addressing Mode

Floating Point Coprocessor Only		
Data Size	Load	Store
Word	MIPS64 MIPS32 Release 2	MIPS64 MIPS32 Release 2
Doubleword	MIPS64 MIPS32 Release 2	MIPS64 MIPS32 Release 2
Unaligned Doubleword Indexed	MIPS64 MIPS32 Release 2	MIPS64 MIPS32 Release 2

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
- Doubleword

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 4.3 lists aligned CPU load and store instructions, while unaligned loads and stores are listed in Table 4.4. Each table also lists the MIPS ISA within which an instruction is defined.

Table 4.3 Aligned CPU Load/Store Instructions

Mnemonic	Instruction	Defined in MIPS ISA
LB	Load Byte	MIPS32
LBU	Load Byte Unsigned	MIPS32
LD	Load Doubleword	MIPS64
LH	Load Halfword	MIPS32
LHU	Load Halfword Unsigned	MIPS32
LW	Load Word	MIPS32
LWU	Load Word Unsigned	MIPS64
SB	Store Byte	MIPS32
SD	Store Doubleword	MIPS64
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 4.4. 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 4.4, along with the MIPS ISA within which an instruction is defined.

Table 4.4 Unaligned CPU Load and Store Instructions

Mnemonic	Instruction	Defined in MIPS ISA
LDL	Load Doubleword Left	MIPS64
LDR	Load Doubleword Right	MIPS64
LWL	Load Word Left	MIPS32
LWR	Load Word Right	MIPS32
SDL	Store Doubleword Left	MIPS64
SDR	Store Doubleword Right	MIPS64

Table 4.4 Unaligned CPU Load and Store Instructions (Continued)

Mnemonic	Instruction	Defined in MIPS ISA
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 4.5 lists the LL and SC instructions, along with the MIPS ISA within which an instruction is defined.

Table 4.5 Atomic Update CPU Load and Store Instructions

Mnemonic	Instruction	Defined in MIPS ISA
LL	Load Linked Word	MIPS32
LLD	Load Linked Doubleword	MIPS64
SC	Store Conditional Word	MIPS32
SCD	Store Conditional Doubleword	MIPS64

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 4.6 lists the coprocessor load and store instructions.

Table 4.6 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

Table 4.7 lists the specific FPU load and store instructions;[1] it also lists the MIPS ISA within which an instruction was first defined.

Table 4.7 FPU Load and Store Instructions Using Register + Register Addressing

Mnemonic	Instruction	Defined in MIPS ISA
LWXC1	Load Word Indexed to Floating Point	MIPS64 MIPS32 Release 2
SWXC1	Store Word Indexed from Floating Point	MIPS64 MIPS32 Release 2
LDXC1	Load Doubleword Indexed to Floating Point	MIPS64 MIPS32 Release 2
SDXC1	Store Doubleword Indexed from Floating Point	MIPS64 MIPS32 Release 2
LUXC1	Load Doubleword Indexed Unaligned to Floating Point	MIPS64 MIPS32 Release 2
SUXC1	Store Doubleword Indexed Unaligned from Floating Point	MIPS64 MIPS32 Release 2

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.

[1] FPU loads and stores are listed here with the other coprocessor loads and stores for convenience.

MIPS32 provided 32-bit integers and 32-bit arithmetic. MIPS64 adds 64-bit integers and provides separate arithmetic and shift instructions for 64-bit operands.

4.1.2.1 ALU Immediate and Three-Operand Instructions

Table 4.8 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 4.8 ALU Instructions With a 16-bit Immediate Operand

Mnemonic	Instruction	Defined in MIPS ISA
ADDI	Add Immediate Word	MIPS32
ADDIU[1]	Add Immediate Unsigned Word	MIPS32
ANDI	And Immediate	MIPS32
DADDI	Doubleword Add Immediate	MIPS64
DADDIU[1]	Doubleword Add Immediate Unsigned	MIPS64
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

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

Table 4.9 describes ALU instructions that use three operands, along with the MIPS ISA within which an instruction is defined.

Table 4.9 Three-Operand ALU Instructions

Mnemonic	Instruction	Defined in MIPS ISA
ADD	Add Word	MIPS32
ADDU[1]	Add Unsigned Word	MIPS32
AND	And	MIPS32
DADD	Doubleword Add	MIPS64
DADDU[1]	Doubleword Add Unsigned	MIPS64
DSUB	Doubleword Subtract	MIPS64

Table 4.9 Three-Operand ALU Instructions (Continued)

Mnemonic	Instruction	Defined in MIPS ISA
DSUBU[1]	Doubleword Subtract Unsigned	MIPS64
NOR	Nor	MIPS32
OR	Or	MIPS32
SLT	Set on Less Than	MIPS32
SLTU	Set on Less Than Unsigned	MIPS32
SUB	Subtract Word	MIPS32
SUBU[1]	Subtract Unsigned Word	MIPS32
XOR	Exclusive Or	MIPS32

[1] 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 4.9 describes ALU instructions that use two operands, along with the MIPS ISA within which an instruction is defined.

Table 4.10 Two-Operand ALU Instructions

Mnemonic	Instruction	Defined in MIPS ISA
CLO	Count Leading Ones in Word	MIPS32
CLZ	Count Leading Zeros in Word	MIPS32
DCLO	Count Leading Ones in Doubleword	MIPS64
DCLZ	Count Leading Zeros in Doubleword	MIPS64

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)

The instructions with a fixed shift amount are limited to a 5-bit shift count, so there are separate instructions for doubleword shifts of 0-31 bits (for instance, DSLL) and 32-63 bits (for instance, DSLL32).

Shift instructions are listed in Table 4.11, along with the MIPS ISA within which an instruction is defined.

Table 4.11 Shift Instructions

Mnemonic	Instruction	Defined in MIPS ISA
DROTR	Doubleword Rotate Right	MIPS64 Release 2

Table 4.11 Shift Instructions (Continued)

Mnemonic	Instruction	Defined in MIPS ISA
DROTR32	Doubleword Rotate Right Plus 32	MIPS64 Release 2
DROTRV	Doubleword Rotate Right Variable	MIPS64 Release 2
DSLL	Doubleword Shift Left Logical	MIPS64
DSLL32	Doubleword Shift Left Logical + 32	MIPS64
DSLLV	Doubleword Shift Left Logical Variable	MIPS64
DSRA	Doubleword Shift Right Arithmetic	MIPS64
DSRA32	Doubleword Shift Right Arithmetic + 32	MIPS64
DSRAV	Doubleword Shift Right Arithmetic Variable	MIPS64
DSRL	Doubleword Shift Right Logical	MIPS64
DSRL32	Doubleword Shift Right Logical + 32	MIPS64
DSRLV	Doubleword Shift Right Logical Variable	MIPS64
ROTR	Rotate Word Right	MIPS32 Release 2
ROTRV	Rotate Word Right Variable	MIPS32 Release 2
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 4.12 lists the multiply, divide, and *HI*/*LO* move instructions, along with the MIPS ISA within which an instruction is defined.

Table 4.12 Multiply/Divide Instructions

Mnemonic	Instruction	Defined in MIPS ISA
DDIV	Doubleword Divide	MIPS64
DDIVU	Doubleword Divide Unsigned	MIPS64
DIV	Divide Word	MIPS32
DIVU	Divide Unsigned Word	MIPS32
DMULT	Doubleword Multiply	MIPS64
DMULTU	Doubleword Multiply Unsigned	MIPS64
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
- Delay Slot Behavior
- 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 branchis 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 **UNPREDICTABLE**.

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 Delay Slot Behavior

There are two versions of branches and jumps; 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** and **Jump** 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 4.13 lists instructions that jump to a procedure call within the current 256 MB-aligned region.

Table 4.14 lists instructions that jump to an absolute address held in a register.

Table 4.13 lists the unconditional jump instructions within a given 256 MByte region. Table 4.15 lists branch instructions that compare two registers before conditionally executing a PC-relative branch. Table 4.16 lists branch instructions that test a register—compare with zero—before conditionally executing a PC-relative branch. Table 4.17 lists the deprecated Branch Likely Instructions.

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

Table 4.13 Unconditional Jump Within a 256 Megabyte Region

Mnemonic	Instruction	Defined in MIPS ISA
J	Jump	MIPS32
JAL	Jump and Link	MIPS32
JALX	Jump and Link Exchange	MIPS16e MIPS32 Release 3

Table 4.14 Unconditional Jump using Absolute Address

Mnemonic	Instruction	Defined in MIPS ISA
JALR	Jump and Link Register	MIPS32
JALR.HB	Jump and Link Register with Hazard Barrier	MIPS32 Release 2
JR	Jump Register	MIPS32
JR.HB	Jump Register with Hazard Barrier	MIPS32 Release 2

Table 4.15 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 4.16 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 4.17 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 and SYNCI)
- Exception Instructions
- Conditional Move Instructions
- Prefetch Instructions
- NOP Instructions

4.1.4.1 Instruction Serialization (SYNC and SYNCI)

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.

The SYNCI instruction synchronizes the processor caches with previous writes or other modifications to the instruction stream.

Table 4.18 lists the synchronization instructions, along with the MIPS ISA within which it is defined.

Table 4.18 Serialization Instruction

Mnemonic	Instruction	Defined in MIPS ISA
SYNC	Synchronize Shared Memory	MIPS32
SYNCI	Synchronize Caches to Make Instruction Writes Effective	MIPS32 Release 2

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 4.19 lists the system call and breakpoint instructions. Table 4.20 lists the trap instructions that compare two registers. Table 4.21 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 4.19 System Call and Breakpoint Instructions

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

Table 4.20 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	MIPS32
TNE	Trap if Not Equal	MIPS32

Table 4.21 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 4.22 lists conditional move instructions, along with the MIPS ISA within which an instruction is defined.

Table 4.22 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

Table 4.22 CPU Conditional Move Instructions

Mnemonic	Instruction	Defined in MIPS ISA
MOVZ	Move Conditional on Zero	MIPS32

4.1.4.4 Prefetch Instructions

There are two prefetch advisory instructions:

- One with register+offset addressing (PREF)
- One with register+register addressing (PREFX)

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. The PREFX instruction is encoded in the FPU *opcode* space, along with the other operations using register+register addressing

Table 4.23 Prefetch Instructions

Mnemonic	Instruction	Addressing Mode	Defined in MIPS ISA
PREF	Prefetch	Register+Offset	MIPS32
PREFX	Prefetch Indexed	Register+Register	MIPS64 MIPS32 Release 2

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 4.24 lists conditional move instructions, along with the MIPS ISA within which an instruction is defined.

Table 4.24 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 4.25.

Table 4.25 Coprocessor Definition and Use in the MIPS Architecture

Coprocessor	MIPS32	MIPS64
CP0	Sys Control	Sys Control
CP1	FPU	FPU
CP2	implementation specific	
CP3		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[2]), 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[3]). In Release 1 of the MIPS64 Architecture, and in Release 2 of the MIPS32 and MIPS64 Architectures, the FPU also uses the computation *opcode* space assigned to coprocessor unit 3, renamed COP1X. Details of the FPU instructions are documented in "Overview of the FPU Instruction Set" on page 69.

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

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

[3] FPU instructions (such as LWC1, SDC1, etc.) that use the COP1 opcode are differentiated from the CP1 designation in this book. See "Overview of the FPU Instruction Set" on page 69 for more information about the FPU instructions.

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 54.

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)

Table 4.26 describes the fields used in these instructions.

Table 4.26 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

4.2.1 CPU Instruction Restrictions

Most 32-bit integer CPU instructions (aside from shifts) require properly sign-extended 32-bit integer operands for well-defined behavior.

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				

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

Overview of the FPU Instruction Set

This chapter describes the instruction set architecture (ISA) for the floating point unit (FPU) in the MIPS64 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:

- “Binary Compatibility” on page 69
- “Enabling the Floating Point Coprocessor” on page 70
- “IEEE Standard 754” on page 70
- “FPU Data Types” on page 70
- “Floating Point Register Types” on page 75
- “Floating Point Control Registers (FCRs)” on page 77
- “Formats of Values Used in FP Registers” on page 84
- “FPU Exceptions” on page 84
- “FPU Instructions” on page 88
- “Valid Operands for FPU Instructions” on page 95
- “FPU Instruction Formats” on page 97

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. In Release 1 of the Architecture, 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. In Release 2

[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://grouper.ieee.org/groups/754/.

of the Architecture and subsequent releases, a 32-bit CPU may include a full 64-bit coprocessor, including a floating point unit which implements the same mode bit to select 32-bit or 64-bit FPU register model.

Any processor implementing MIPS64 can also run MIPS32 binary programs, built for the same, or a lower release of the Architecture, 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 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)
- 64-bit **paired single** floating point, combining two single precision data types (Type PS, shown in Figure 5-3)

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 (or each half of Paired Single)	Double
Bits of mantissa precision, p	24	53
Maximum exponent, E_max	+127	+1023
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, 5-2, and 5-3 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)

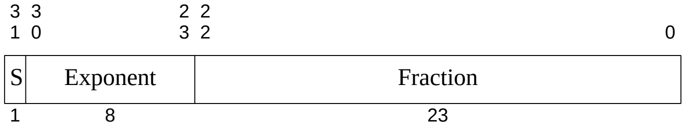


Figure 5-2 Double-Precisions Floating Point Format (D)

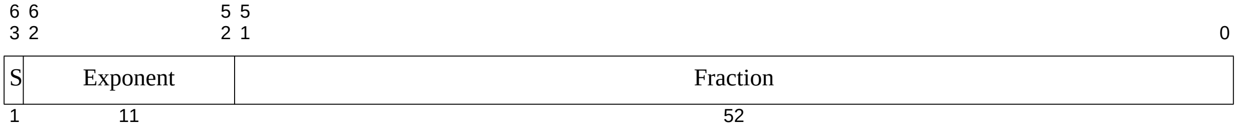
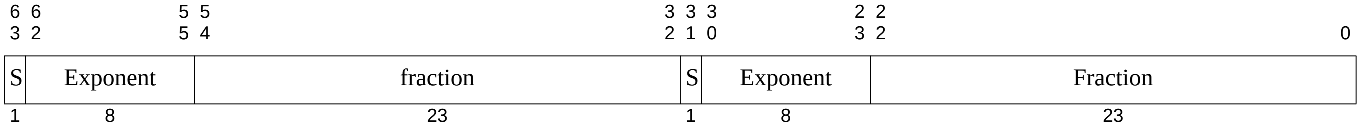


Figure 5-3 Paired Single Floating Point Format (PS)



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[1]	Typical Double Bit Pattern[1]
E_max + 1	≠ 0		1	SNaN	Signaling NaN	`0x7fffffff`	`0x7fffffff ffffffff`
			0	QNaN	Quiet NaN	`0x7fbfffff`	`0x7ff7ffff ffffffff`
E_max +1	0	1		- ∞	minus infinity	`0xff800000`	`0xfff00000 00000000`
		0		+ ∞	plus infinity	`0x7f800000`	`0x7ff00000 00000000`

E_max
to
E_min
		1		- $(2^E)(1.f)$	negative normalized number	`0x80800000`
						through
						`0xff7fffff`
							`0x80100000 00000000`
							through
							`0xffefffff ffffffff`

		0		+ $(2^E)(1.f)$	positive normalized number	`0x00800000`
						through
						`0x7f7fffff`
							`0x00100000 00000000`
							through
							`0x7fefffff ffffffff`

E_min -1	≠ 0	1		- $(2^{E\_min})(0.f)$	negative denormalized number	`0x807fffff`	`0x800fffff ffffffff`
		0		+ $(2^{E\_min})(0.f)$	positive denormalized number	`0x007fffff`	`0x000fffff ffffffff`
E_min -1	0	1		- 0	negative zero	`0x80000000`	`0x80000000 00000000`
		0		+ 0	positive zero	`0x00000000`	`0x00000000 00000000`

[1] 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 5.8.1 "Exception Conditions" on page 85.

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 86).

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	`0x7fbf ffff`
Double floating point	`0x7ff7 ffff ffff ffff`
Word fixed point	`0x7fff ffff`
Longword fixed point	`0x7fff ffff ffff ffff`

5.4.1.6 Paired Single Exceptions

Exception conditions that arise while executing the two halves of a floating point vector operation are ORed together, and the instruction is treated as having caused all the exceptional conditions arising from both operations. The hardware makes no effort to determine which of the two operations encountered the exceptional condition.

5.4.1.7 Paired Single Condition Codes

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

5.4.2 Fixed Point Formats

The FPU provides two fixed point data types:

- 32-bit **Word** fixed point (type *W*), shown in Figure 5-4
- 64-bit **Longword** fixed point (type *L*), shown in Figure 5-5

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 5-4 Word Fixed Point Format (W)

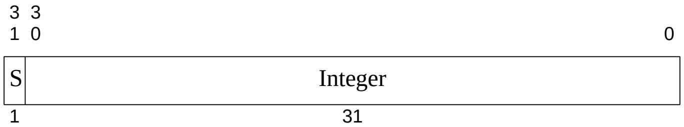
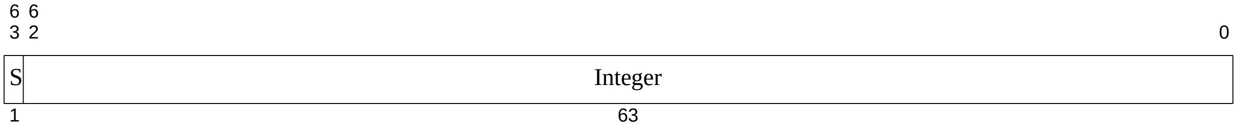


Figure 5-5 Longword Fixed Point Format (L)



5.5 Floating Point Register Types

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

In Release 1 of the Architecture, 64-bit floating point units were supported only by implementations of the MIPS64 Architecture. Similarly, implementations of MIPS32 of the Architecture only supported 32-bit floating point units. In Release 2 of the Architecture and MIPSr3, a 64-bit floating point unit is supported on implementations of both the MIPS32 and MIPS64 Architectures.

Floating Point registers (*FPR*s) are 32 or 64 bits wide. A 32-bit floating point unit contains 32 32-bit FPRs, each of which is capable of storing a 32-bit data type. Double-precision (type D) data types are stored in even-odd pairs of FPRs, and the long-integer (type L) and paired single (type PS) data types are not supported. A 64-bit floating point unit contains 32 64-bit FPRs, each of which is capable of storing any data type. For compatibility with 32-bit FPUs, the FR bit in the CP0 *Status* register is used by a MIPS64 Release 1, or any Release 2 or subsequent releases processor that supports a 64-bit FPU to configure the FPU in a mode in which the FPRs are treated as 32 32-bit registers, each of which is capable of storing only 32-bit data types. In this mode, the double-precision floating point (type D) data type is stored in even-odd pairs of FPRs, and the long-integer (type L) and paired single (type PS) data types are not supported.

- 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 FPU Register Models

There are separate FPU register models in Release 1 of the Architecture:

- MIPS32 defines 32 32-bit registers, with D-format values stored in even-odd pairs of registers.

- MIPS64 defines 32 64-bit registers, with all formats supported in a register.

To support MIPS32 programs, MIPS64 processors also provide the MIPS32 register model, which is available as a mode selection through the **FR Bit of the CP0 Status Register**.

If the value of FR bit is changed, the contents of the FPRs becomes **UNPREDICTABLE**. For some implementations, it might be necessary for software to re-initialize the FPRs.

In Release 2 of the Architecture and subsequent releases, both FPU register models are available for implementations, and the FR bit of the CP0 Status Register.

5.5.2 Binary Data Transfers (32-Bit and 64-Bit)

The data transfer instructions move words and doublewords between the FPU FPRs and the remainder of the system. The operations of the word and doubleword load and move-to instructions are shown in Figure 5-6 and Figure 5-7.

The store and move-from instructions operate in reverse, reading data from the location which the corresponding load or move-to instruction wrote.

Figure 5-6 FPU Word Load and Move-to Operations

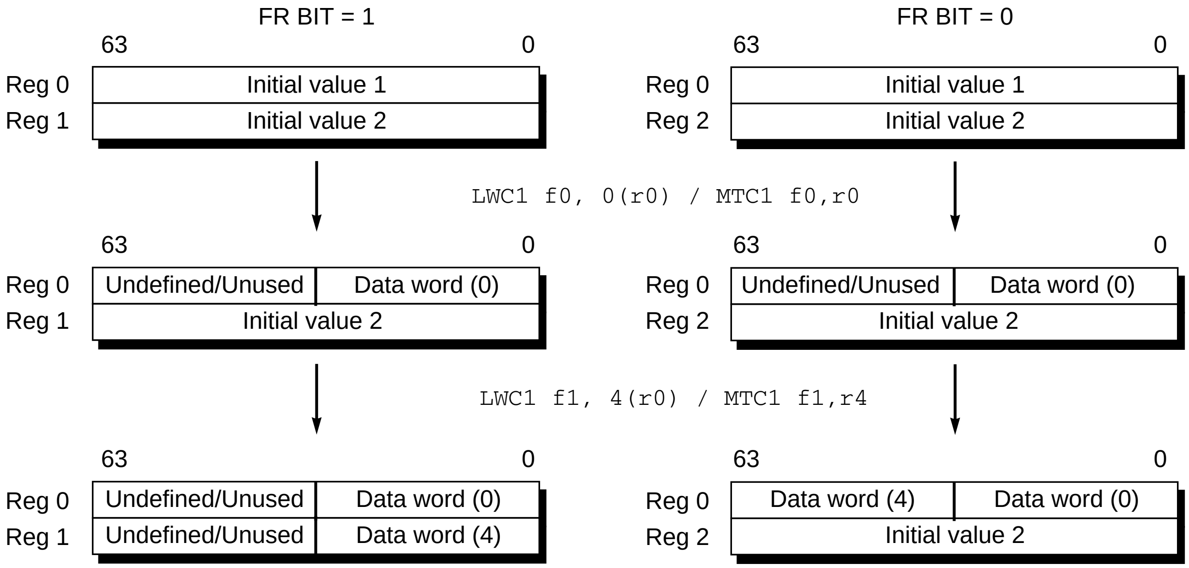
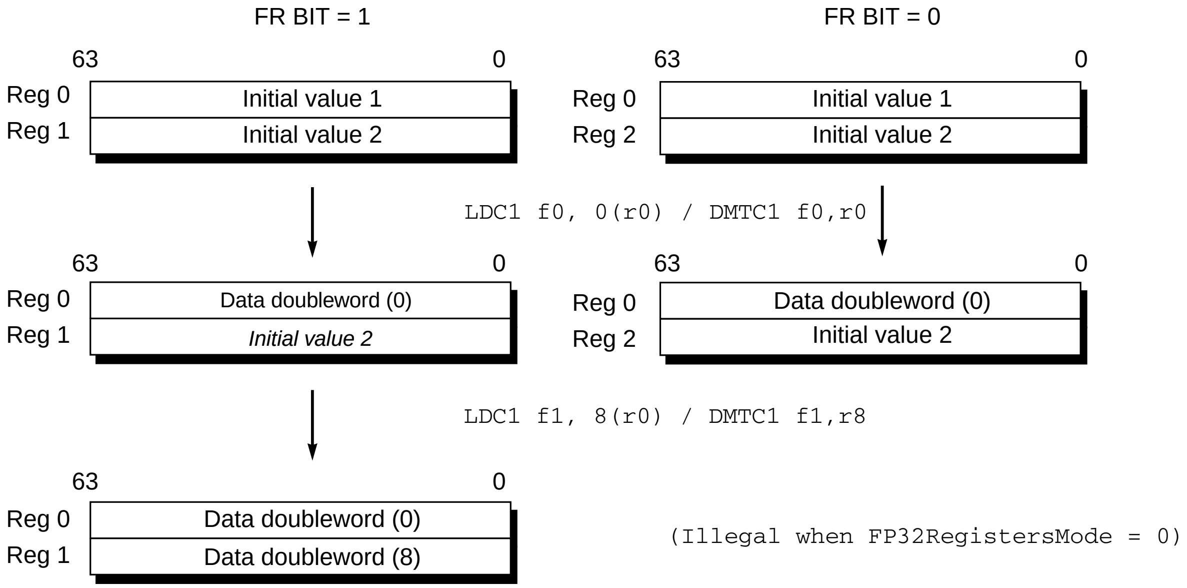


Figure 5-7 FPU Doubleword Load and Move-to Operations



5.5.3 FPRs and Formatted Operand Layout

FPU instructions that operate on formatted operand values specify the **floating point register** (FPR) that holds the value. Operands that are only 32 bits wide (*W* and *S* formats), use only half the space in a 64-bit FPR.

The FPR organization and the way that operand data is stored in them is shown in Figures 5-8, 5-9 and 5-10.

Figure 5-8 Single Floating Point or Word Fixed Point Operand in an FPR

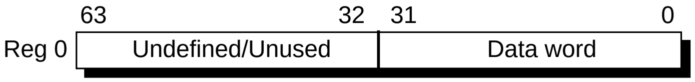


Figure 5-9 Double Floating Point or Longword Fixed Point Operand in an FPR

	63 … 0
Reg 0	Data doubleword/Longword

Figure 5-10 Paired-Single Floating Point Operand in an FPR

	63 … 32	31 … 0
Reg 0	Paired-Single	Paired-Single

5.6 Floating Point Control Registers (FCRs)

The MIPS64 Architecture supports the following five floating point *Control* registers (*FCR*s):

- *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 (FIR, 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 5-11 shows the format of the *FIR* register; Table 5.4 describes the *FIR* register fields.

Figure 5-11 FIR Register Format

31 … 28	27 … 24	23	22	21	20	19	18	17	16	15 … 8	7 … 0
0 0000	Impl	0	F64	L	W	3D	PS	D	S	ProcessorID	Revision

Table 5.4 FIR Register Field Descriptions

Fields					
Name	**Bits**	**Description**	**Read/ Write**	**Reset State**	**Compliance**
0	31:28	Reserved for future use; reads as zero	0	0	Reserved

Table 5.4 FIR Register Field Descriptions (Continued)

<table>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read/ Write</th><th rowspan="2">Reset State</th><th rowspan="2">Compliance</th></tr>
<tr><th>Name</th><th>Bits</th></tr>
<tr><td>Impl</td><td>27..24</td><td>These bits are implementation dependent and are not defined by the architecture, other than the fact that they are read-only. This bits are explicitly not intended to be used for mode control functions.</td><td>R</td><td>Preset</td><td>Optional</td></tr>
<tr><td>0</td><td>23</td><td>Reserved for future use; reads as zero</td><td>0</td><td>0</td><td>Reserved</td></tr>
<tr><td>F64</td><td>22</td><td>Indicates that the floating point unit has registers and data paths that are 64-bits wide. This bit was added in Release 2 of the Architecture, and is a one on either any processors with a 64-bit floating point unit, and a zero on any processors with a 32-bit floating point unit. A value of one in this bit indicates that $Status_{FR}$ is implemented.
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>FPU is 32 bits</td></tr>
<tr><td>1</td><td>FPU is 64 bits</td></tr>
</table></td><td>R</td><td>Preset</td><td>Required (Release 2)</td></tr>
<tr><td>L</td><td>21</td><td>Indicates that the longword fixed point (L) data type and instructions are implemented:
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>L fixed point not implemented</td></tr>
<tr><td>1</td><td>L fixed point implemented</td></tr>
</table></td><td>R</td><td>Preset</td><td>Required (Release 2)</td></tr>
<tr><td>W</td><td>20</td><td>Indicates that the word fixed point (W) data type and instructions are implemented:
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>W fixed point not implemented</td></tr>
<tr><td>1</td><td>W fixed point implemented</td></tr>
</table></td><td>R</td><td>Preset or Externally Set</td><td>Required (Release 2)</td></tr>
</table>

Table 5.4 FIR Register Field Descriptions (Continued)

<table>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read/ Write</th><th rowspan="2">Reset State</th><th rowspan="2">Compliance</th></tr>
<tr><th>Name</th><th>Bits</th></tr>
<tr><td>3D</td><td>19</td><td>the MIPS-3D ASE is supported on any processors with a 64-bit floating point unit, and this bit indicates that the MIPS-3D ASE is implemented:
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>MIPS-3D ASE not implemented</td></tr>
<tr><td>1</td><td>MIPS-3D ASE implemented</td></tr>
</table>
Indicates that the MIPS-3D ASE is implemented:
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>MIPS-3D ASE not implemented</td></tr>
<tr><td>1</td><td>MIPS-3D ASE implemented</td></tr>
</table></td><td>R</td><td>Preset</td><td>Required</td></tr>
<tr><td>PS</td><td>18</td><td>tIndicates that the paired single floating point data type is implemented:
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>PS floating point not implemented</td></tr>
<tr><td>1</td><td>PS floating point implemented</td></tr>
</table></td><td>R</td><td>Preset</td><td>Required</td></tr>
<tr><td>D</td><td>17</td><td>Indicates that the double-precision (D) floating point data type and instructions are implemented:
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>D floating point not implemented</td></tr>
<tr><td>1</td><td>D floating point implemented</td></tr>
</table></td><td>R</td><td>Preset</td><td>Required</td></tr>
<tr><td>S</td><td>16</td><td>Indicates that the single-precision (S) floating point data type and instructions are implemented:
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>S floating point not implemented</td></tr>
<tr><td>1</td><td>S floating point implemented</td></tr>
</table></td><td>R</td><td>Preset</td><td>Required</td></tr>
<tr><td>ProcessorID</td><td>15:8</td><td>Identifies the floating point processor.</td><td>R</td><td>Preset</td><td>Required</td></tr>
<tr><td>Revision</td><td>7:0</td><td>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.</td><td>R</td><td>Preset</td><td>Optional</td></tr>
</table>

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 5-12 shows the format of the *FCSR* register; Table 5.5 describes the *FCSR* register fields.

Figure 5-12 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

Table 5.5 FCSR Register Field Descriptions (Continued)

Fields		Description	Read/ Write	Reset State	Compliance
Name	Bits				
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
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.
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 5-13 shows the format of the *FCCR* register; Table 5.8 describes the *FCCR* register fields.

Figure 5-13 FCCR Register Format

31 8	7 0
0 0000 0000 0000 0000 0000 0000	FCC
	7 6 5 4 3 2 1 0

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 5-14 shows the format of the *FEXR* register; Table 5.9 describes the *FEXR* register fields.

Figure 5-14 FEXR Register Format

31 18	17	16	15	14	13	12	11 7	6	5	4	3	2	1 0
0 0000 0000 0000 00	Cause						0 00 000	Flags					0 00
	E	V	Z	O	U	I		V	Z	O	U	I	

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 5-15 shows the format of the *FENR* register; Table 5.10 describes the *FENR* register fields.

Figure 5-15 FENR Register Format

31 12	11 10 9 8 7	6 3	2	1 0
0 0000 0000 0000 0000 0000	Enables	0 000 0	FS	RM

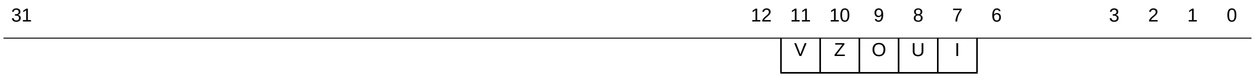


Table 5.10 FENR Register Field Descriptions

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 contents of the floating point operand register is interpreted as the format defined by the instruction which is being executed. That is, there is no persistent interpretation of the register values.

5.8 FPU Exceptions

This section provides the following information FPU exceptions:

- Precise exception mode
- Descriptions of the exceptions
- Non-Arithmetic Instructions

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: $0 \times \infty$, 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 ±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 ±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 ±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 ±$2^{\mathrm{E\_min}}$.

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.

5.8.1.7 Non-Arithmetic Instructions

Some FPU conversion and FPU Formatted Operand-Value Move instructions (see next section) do not perform floating-point arithmetic operations on their input operands. For that reason, such instructions do not generate IEEE arithmetic exceptions. These instructions include MOV.fmt, MOVF.fmt, MOVT.fmt, MOVZ.fmt, MOVN.fmt, PLL.PS, PLU.PS, PUL.PS, PUU.PS, CVT.S.PU, CVT.PS.S, CVT.S.PL.

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/doubleword 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 37).

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. Moreover, for the FPU only, there are load and store instructions using *register+register* addressing.

Tables 5.13 through 5.15 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
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 Loads and Using Register+Register Address Mode

Mnemonic	Instruction	Defined in MIPS ISA
LDXC1	Load Doubleword Indexed to Floating Point	MIPS64 MIPS32 Release 2
LUXC1	Load Doubleword Indexed Unaligned to Floating Point	MIPS64 MIPS32 Release 2
LWXC1	Load Word Indexed to Floating Point	MIPS64 MIPS32 Release 2
SDXC1	Store Doubleword Indexed to Floating Point	MIPS64 MIPS32 Release 2
SUXC1	Store Doubleword Indexed Unaligned to Floating Point	MIPS64 MIPS32 Release 2

Table 5.14 FPU Loads and Using Register+Register Address Mode (Continued)

Mnemonic	Instruction	Defined in MIPS ISA
SWXC1	Store Word Indexed to Floating Point	MIPS64 MIPS32 Release 2

Table 5.15 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
DMFC1	Doubleword Move From Floating Point	MIPS64
DMTC1	Doubleword Move To Floating Point	MIPS64
MFC1	Move Word From Floating Point	MIPS32
MFHC1	Move Word from High Half of Floating Point Register	MIPS32 Release 2
MTC1	Move Word To Floating Point	MIPS32
MTHC1	Move Word to High Half of Floating Point Register	MIPS32 Release 2

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 5.16.

Table 5.16 FPU IEEE Arithmetic Operations

Mnemonic	Instruction	Defined in MIPS ISA
ABS.fmt	Floating Point Absolute Value	MIPS32
ABS.fmt (*PS*)	Floating Point Absolute Value (Paired Single)	MIPS64 MIPS32 Release 2
ADD.fmt	Floating Point Add	MIPS32
ADD.fmt (*PS*)	Floating Point Add (Paired Single)	MIPS64 MIPS32 Release 2
C.cond.fmt	Floating Point Compare	MIPS32
C.cond.fmt (*PS*)	Floating Point Compare (Paired Single)	MIPS64 MIPS32 Release 2
DIV.fmt	Floating Point Divide	MIPS32
MUL.fmt	Floating Point Multiply	MIPS32

Table 5.16 FPU IEEE Arithmetic Operations (Continued)

Mnemonic	Instruction	Defined in MIPS ISA
MUL.fmt (*PS*)	Floating Point Multiply (Paired Single)	MIPS64 MIPS32 Release 2
NEG.fmt	Floating Point Negate	MIPS32
NEG.fmt (*PS*)	Floating Point Negate (Paired Single)	MIPS64 MIPS32 Release 2
SQRT.fmt	Floating Point Square Root	MIPS32
SUB.fmt	Floating Point Subtract	MIPS32
SUB.fmt (*PS*)	Floating Point Subtract (Paired Single)	MIPS64 MIPS32 Release 2

Two operations, Reciprocal Approximation (RECIP) and Reciprocal Square Root Approximation (RSQRT), may be less accurate than the IEEE specification:

- The result of RECIP differs from the exact reciprocal by no more than one ULP.
- The result of RSQRT differs from the exact reciprocal square root by no more than two ULPs.

Within these error limits, the results of these instructions are implementation specific.

A list of FPU-approximate arithmetic operations is given in Table 5.17..

Table 5.17 FPU-Approximate Arithmetic Operations

Mnemonic	Instruction	Defined in MIPS ISA
RECIP.fmt	Floating Point Reciprocal Approximation	MIPS64 MIPS32 Release 2
RSQRT.fmt	Floating Point Reciprocal Square Root Approximation	MIPS64 MIPS32 Release 2

Four compound-operation instructions perform variations of multiply-accumulate—that is, multiply two operands, accumulate the result to a third operand, and produce a result. These instructions are listed in Table 5.18. The product is rounded according to the current rounding mode prior to the accumulation. The accumulated result is also rounded. This model meets the IEEE-754-1985 accuracy specification; the result is numerically identical to an equivalent computation using a sequence of multiply, add/subtract, or negate instructions. Similarly, exceptions and flags behave as if the operation was implemented with a sequence of multiply, add/subtract and negate instructions. This behavior is often known as “Non-Fused”.

Table 5.18 lists the FPU Multiply-Accumulate arithmetic operations.

Table 5.18 FPU Multiply-Accumulate Arithmetic Operations

Mnemonic	Instruction	Defined in MIPS ISA
MADD.fmt	Floating Point Multiply Add	MIPS64 MIPS32 Release 2

Table 5.18 FPU Multiply-Accumulate Arithmetic Operations

Mnemonic	Instruction	Defined in MIPS ISA
MADD.fmt (*PS*)	Floating Point Multiply Add (Paired Single)	MIPS64 MIPS32 Release 2
MSUB.fmt	Floating Point Multiply Subtract	MIPS64 MIPS32 Release 2
MSUB.fmt (*PS*)	Floating Point Multiply Subtract (Paired Single)	MIPS64 MIPS32 Release 2
NMADD.fmt	Floating Point Negative Multiply Add	MIPS64 MIPS32 Release 2
NMADD.fmt (*PS*)	Floating Point Negative Multiply Add (Paired Single)	MIPS64 MIPS32 Release 2
NMSUB.fmt	Floating Point Negative Multiply Subtract	MIPS64 MIPS32 Release 2
NMSUB.fmt (*PS*)	Floating Point Negative Multiply Subtract (Paired Single)	MIPS64 MIPS32 Release 2

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. Tables 5.19 and 5.20 list the FPU conversion instructions according to their rounding mode.

Table 5.19 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.L.fmt	Floating Point Convert to Long Fixed Point	MIPS64 MIPS32 Release 2
CVT.PS.*S*	Floating Point Convert Pair to Paired Single	MIPS64 MIPS32 Release 2
CVT.S.fmt	Floating Point Convert to Single Floating Point	MIPS32
CVT.S.fmt (*PL*, *PU*)	Floating Point Convert to Single Floating Point (Paired Lower, Paired Upper)	MIPS64 MIPS32 Release 2
CVT.W.fmt	Floating Point Convert to Word Fixed Point	MIPS32

Table 5.20 FPU Conversion Operations Using a Directed Rounding Mode

Mnemonic	Instruction	Defined in MIPS ISA
CEIL.L.fmt	Floating Point Ceiling to Long Fixed Point	MIPS64 MIPS32 Release 2

Table 5.20 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.L.fmt	Floating Point Floor to Long Fixed Point	MIPS64 MIPS32 Release 2
FLOOR.W.fmt	Floating Point Floor to Word Fixed Point	MIPS32
ROUND.L.fmt	Floating Point Round to Long Fixed Point	MIPS64 MIPS32 Release 2
ROUND.W.fmt	Floating Point Round to Word Fixed Point	MIPS32
TRUNC.L.fmt	Floating Point Truncate to Long Fixed Point	MIPS64 MIPS32 Release 2
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 **UNPREDICTABLE**. (For more information, see the individual descriptions of the conditional move instructions in Volume II.)

These instructions are listed in Tables 5.21 through 5.23.

Table 5.21 FPU Formatted Operand Move Instructions

Mnemonic	Instruction	Defined in MIPS ISA
MOV.fmt	Floating Point Move	MIPS32
MOV.fmt (*PS*)	Floating Point Move (Paired Single)	MIPS64 MIPS32 Release 2

Table 5.22 FPU Conditional Move on True/False Instructions

Mnemonic	Instruction	Defined in MIPS ISA
MOVF.fmt	Floating Point Move Conditional on FP False	MIPS32
MOVF.fmt (*PS*)	Floating Point Move Conditional on FP False (Paired Single)	MIPS64 MIPS32 Release 2
MOVT.fmt	Floating Point Move Conditional on FP True	MIPS32

Table 5.22 FPU Conditional Move on True/False Instructions

Mnemonic	Instruction	Defined in MIPS ISA
MOVT.fmt (*PS*)	Floating Point Move Conditional on FP True (Paired Single)	MIPS64 MIPS32 Release 2

Table 5.23 FPU Conditional Move on Zero/Nonzero Instructions

Mnemonic	Instruction	Defined in MIPS ISA
MOVN.fmt	Floating Point Move Conditional on Nonzero	MIPS32
MOVN.fmt (*PS*)	Floating Point Move Conditional on Nonzero (Paired Single)	MIPS64 MIPS32 Release 2
MOVZ.fmt	Floating Point Move Conditional on Zero	MIPS32
MOVZ.fmt (*PS*)	Floating Point Move Conditional on Zero (Paired Single)	MIPS64 MIPS32 Release 2

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 MIPS64 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 5.24 lists the conditional branch FPU instructions; Table 5.25 lists the deprecated conditional branch likely instructions.

Table 5.24 FPU Conditional Branch Instructions

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

Table 5.24 FPU Conditional Branch Instructions

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

Table 5.25 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. It also defines an instruction to align a misaligned pair of paired-single values (ALNV.PS) and a quartet of instructions that merge a pair of paired-single values (PLL.PS, PLU.PS, PUL.PS, PUU.PS).

Table 5.26 lists these conditional move instructions.

Table 5.26 CPU Conditional Move on FPU True/False Instructions

Mnemonic	Instruction	Defined in MIPS ISA
ALNV.PS	FP Align Variable	MIPS64 MIPS32 Release 2
MOVN	Move Conditional on FP False	MIPS32
MOVZ	Move Conditional on FP True	MIPS32
PLL.PS	Pair Lower Lower	MIPS64 MIPS32 Release 2
PLU.PS	Pair Lower Upper	MIPS64 MIPS32 Release 2
PUL.PS	Pair Upper Lower	MIPS64 MIPS32 Release 2
PUU.PS	Pair Upper Upper	MIPS64 MIPS32 Release 2

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* or *fmt3* 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* and *fmt3* field are shown in Table 5.27.

Table 5.27 FPU Operand Format Field (*fmt*, *fmt3*) Encoding

fmt	fmt3	Instruction Mnemonic	Size		Data Type
			Name	Bits	
0-15	-	Reserved			
16	0	S	single	32	Floating point
17	1	D	double	64	Floating point
18-19	2-3	Reserved			
20	4	W	word	32	Fixed point
21	5	L	long	64	Fixed point
22	6	PS	paired single	64 (2x32)	Floating point
23–31	7	Reserved			

The result of an instruction using operand formats marked **U** in Table 5.28 is not currently specified by this architecture and causes a Reserved Instruction exception.

Table 5.28 Valid Formats for FPU Operations

Mnemonic	Operation	Operand Fmt					COP1 Function Value	COP1X op4 Value
		Float			Fixed			
		S	D	PS	W	L		
ABS	Absolute value	•	•	•	**U**	**U**	5	
ADD	Add	•	•	•	**U**	**U**	0	
C.*cond*	Floating Point compare	•	•	•	**U**	**U**	48–63	
CEIL.L, (CEIL.W)	Convert to longword (word) fixed point, round toward +∞	•	•	**U**	**U**	**U**	10 (14)	
CVT.D	Convert to double floating point	•	**U**	**U**	•	•	33	
CVT.L	Convert to longword fixed point	•	•	**U**	**U**	**U**	37	
CVT.S	Convert to single floating point	**U**	•	**U**	•	•	32	
CVT. PU, PL	Convert to single floating point (paired upper, paired lower)	**U**	**U**	•	**U**	**U**	32, 40	
CVT.W	Convert to 32-bit fixed point	•	•	**U**	**U**	**U**	36	
DIV	Divide	•	•	**U**	**U**	**U**	3	
FLOOR.L, (FLOOR.W)	Convert to longword (word) fixed point, round toward –∞	•	•	**U**	**U**	**U**	11 (15)	
MADD	Multiply-Add	•	•	•	**U**	**U**		4

Table 5.28 Valid Formats for FPU Operations (Continued)

Mnemonic	Operation	Operand Fmt					COP1 Function Value	COP1X op4 Value
		Float			Fixed			
		S	D	PS	W	L		
MOV	Move Register	•	•	•	**U**	**U**	6	
MOVC	FP Move conditional on condition	•	•	•	**U**	**U**	17	
MOVN	FP Move conditional on GPR≠zero	•	•	•	**U**	**U**	19	
MOVZ	FP Move conditional on GPR=zero	•	•	•	**U**	**U**	18	
MSUB	Multiply-Subtract	•	•	•	**U**	**U**		5
MUL	Multiply	•	•	•	**U**	**U**	2	
NEG	Negate	•	•	•	**U**	**U**	7	
NMADD	Negative Multiply-Add	•	•	•	**U**	**U**		6
NMSUB	Negative Multiply-Subtract	•	•	•	**U**	**U**		7
PLL, PLU, PUL, PUU	Pair (Lower Lower, Lower Upper, Upper Lower, Upper Upper)	**U**	**U**	•	**U**	**U**	44-47	
RECIP	Reciprocal Approximation	•	•	**U**	**U**	**U**	21	
ROUND.L, (ROUND.W)	Convert to longword (word) fixed point, round to nearest/even	•	•	**U**	**U**	**U**	8 (12)	
RSQRT	Reciprocal square root approximation	•	•	**U**	**U**	**U**	22	
SQRT	Square Root	•	•	**U**	**U**	**U**	4	
SUB	Subtract	•	•	•	**U**	**U**	1	
TRUNC.L, (TRUNC.W)	Convert to longword (word) fixed point, round toward zero	•	•	**U**	**U**	**U**	9 (13)	
Key: • – Valid. **U** – Unimplemented and causes Reserved Instruction Exception.								

5.11 FPU Instruction Formats

An FPU instruction is a single 32-bit aligned word. FP instruction formats are shown in Figures 5-16 through 5-25.

In these figures, variables are labelled in lowercase, such as *offset*. Constants are labelled in uppercase, as are numerals. Following these figures, Table 5.29 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 5-16 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 5-17 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 5-18 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 5-19 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 5-20 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 5-21 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 5-22 Four-Register Formatted Arithmetic FPU Instruction Format

31 26	25 21	20 16	15 11	10 6	5	0
COP1X	fr	ft	fs	fd	op4	fmt3
6	5	5	5	5	3	3

Register-4: Four-register formatted arithmetic operations

Figure 5-23 Register Index FPU Instruction Format

31 26	25 21	20 16	15 11	10 6	5 0
COP1X	base	index	0	fd	function
6	5	5	5	5	6

Register Index: Load and Store using register + register addressing

Figure 5-24 Register Index Hint FPU Instruction Format

31 26	25 21	20 16	15 11	10 6	5 0
COP1X	base	index	hint	0	PREFX
6	5	5	5	5	6

Register Index Hint: Prefetch using register + register addressing

Figure 5-25 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 5.29 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.
COP1X	Coprocessor 1 eXtended 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.
function: op4 + fmt3	*op4* is a 3-bit *function* field specifying a 4-register arithmetic operation for COP1X. *fmt3* is a 3-bit field specifying the format of the operands and destination. The combinations are shown as distinct instructions in the opcode tables.
hint	*Hint* field made available to cache controller for prefetch operation.
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.

Table 5.29 FPU Instruction Format Fields (Continued)

Field	Description
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).
PREFX	Value in *function* field for prefetch instruction when *op*=COP1X.
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.

Appendix A

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.

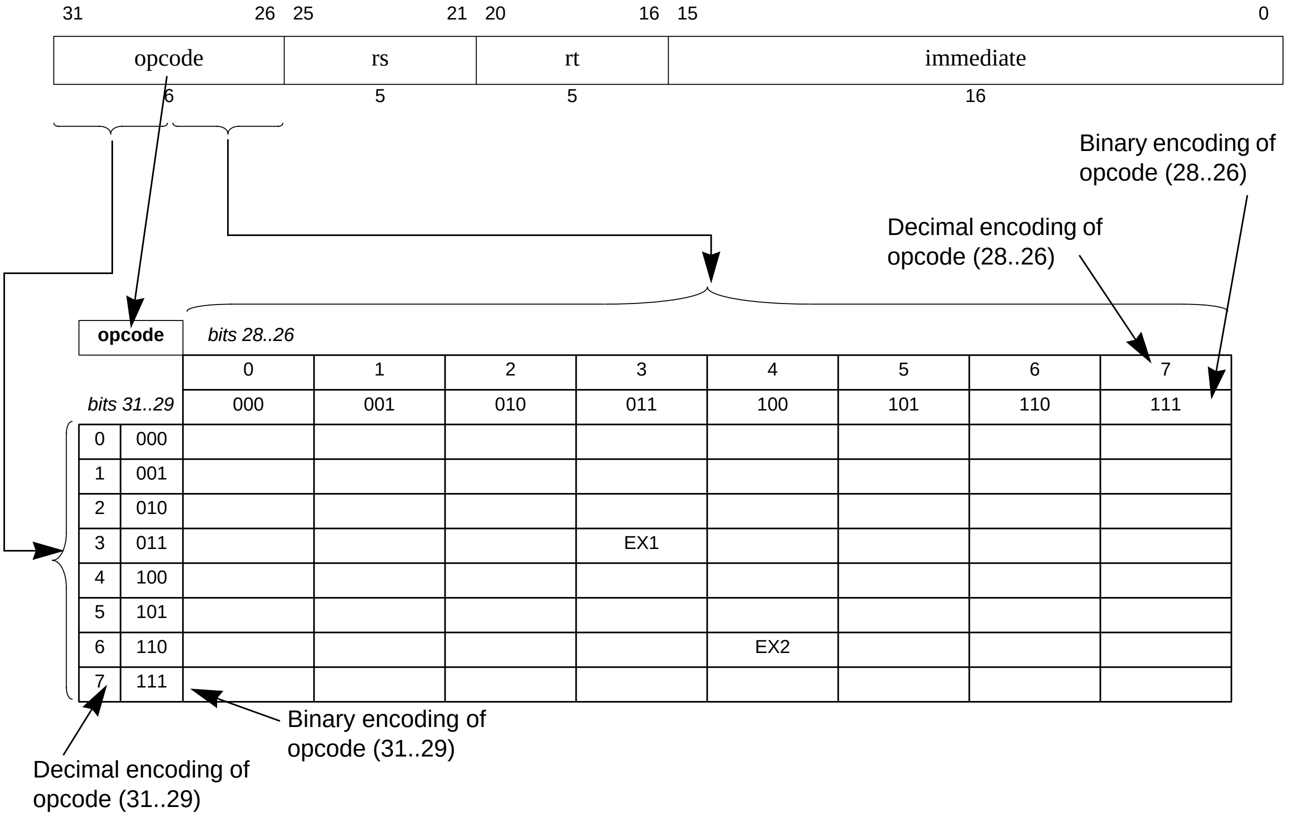
A.2 Instruction Bit Encoding Tables

This section provides various bit encoding tables for the instructions of the MIPS64® 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.23 describe the encoding used for the MIPS64 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 or a new revision of the Architecture. Executing such an instruction must cause a Reserved Instruction Exception.
⊥	Operation or field codes marked with this symbol represent instructions which are not legal if the processor is configured to be backward compatible with MIPS32 processors. If the processor is executing with 64-bit operations enabled, execution proceeds normally. In other cases, executing such an instruction must cause a Reserved Instruction Exception (non-coprocessor encodings or coprocessor instruction encodings for a coprocessor to which access is allowed) or a Coprocessor Unusable Exception (coprocessor instruction encodings for a coprocessor to which access is not allowed).

Table A.1 Symbols Used in the Instruction Encoding Tables (Continued)

Symbol	Meaning
∇	Operation or field codes marked with this symbol represent instructions which were only legal if 64-bit operations were enabled on implementations of Release 1 of the Architecture. In Release 2 of the architecture, operation or field codes marked with this symbol represent instructions which are legal if 64-bit floating point operations are enabled. In other cases, executing such an instruction must cause a Reserved Instruction Exception (non-coprocessor encodings or coprocessor instruction encodings for a coprocessor to which access is allowed) or a Coprocessor Unusable Exception (coprocessor instruction encodings for a coprocessor to which access is not allowed).
θ	Operation or field codes marked with this symbol are available to licensed MIPS partners. To avoid multiple conflicting instruction definitions, MIPS Technologies will assist the partner in selecting appropriate encodings if requested by the partner. The partner is not required to consult with MIPS Technologies 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.
ϕ	Operation or field codes marked with this symbol are obsolete and will be removed from a future revision of the MIPS64 ISA. Software should avoid using these operation or field codes.
⊕	Operation or field codes marked with this symbol are valid for Release 2 implementations of the architecture. Executing such an instruction in a Release 1 implementation must cause a Reserved Instruction Exception.

Table A.2 MIPS64 Encoding of the Opcode Field

opcode		*bits 28..26*							
		0	1	2	3	4	5	6	7
bits 31..29		000	001	010	011	100	101	110	111
0	000	*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	*COP1X* δ	BEQL ϕ	BNEL ϕ	BLEZL ϕ	BGTZL ϕ
3	011	DADDI ⊥	DADDIU ⊥	LDL ⊥	LDR ⊥	*SPECIAL2* δ	JALX ε	MDMX εδ	*SPECIAL3*[1] δ⊕
4	100	LB	LH	LWL	LW	LBU	LHU	LWR	LWU ⊥
5	101	SB	SH	SWL	SW	SDL ⊥	SDR ⊥	SWR	CACHE
6	110	LL	LWC1	LWC2 θ	PREF	LLD ⊥	LDC1	LDC2 θ	LD ⊥
7	111	SC	SWC1	SWC2 θ	∗	SCD ⊥	SDC1	SDC2 θ	SD ⊥

[1] Release 2 of the Architecture added the SPECIAL3 opcode. Implementations of Release 1 of the Architecture signaled a Reserved Instruction Exception for this opcode.

Table A.3 MIPS64 *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[1]	*MOVCI* δ	*SRL* δ	SRA	SLLV	∗	*SRLV* δ	SRAV
1	001	JR[2]	JALR[2]	MOVZ	MOVN	SYSCALL	BREAK	∗	SYNC
2	010	MFHI	MTHI	MFLO	MTLO	DSLLV ⊥	∗	DSRLV δ⊥	DSRAV ⊥
3	011	MULT	MULTU	DIV	DIVU	DMULT ⊥	DMULTU ⊥	DDIV ⊥	DDIVU ⊥
4	100	ADD	ADDU	SUB	SUBU	AND	OR	XOR	NOR
5	101	∗	∗	SLT	SLTU	DADD ⊥	DADDU ⊥	DSUB ⊥	DSUBU ⊥
6	110	TGE	TGEU	TLT	TLTU	TEQ	∗	TNE	∗
7	111	DSLL ⊥	∗	*DSRL* δ⊥	DSRA ⊥	DSLL32 ⊥	∗	*DSRL32* δ⊥	DSRA32 ⊥

[1] Specific encodings of the *rt*, *rd*, and *sa* fields are used to distinguish among the SLL, NOP, SSNOP, EHB and PAUSE functions.

[2] Specific encodings of the *hint* field are used to distinguish JR from JR.HB and JALR from JALR.HB

Table A.4 MIPS64 *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	∗	∗	∗	∗	∗	*	∗	SYNCI ⊕

Table A.5 MIPS64 *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	DCLZ ⊥	DCLO ⊥	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 MIPS64 *SPECIAL3*[1] Encoding of Function Field for Release 2 of the Architecture

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	EXT ⊕	DEXTM ⊥⊕	DEXTU ⊥⊕	DEXT ⊥⊕	INS ⊕	DINSM ⊥⊕	DINSU ⊥⊕	DINS ⊥⊕
1	001	*	*	*	*	*	*	*	*
2	010	*	*	*	*	*	*	*	*
3	011	*	*	*	*	*	*	*	*
4	100	BSHFL ⊕δ	*	*	*	*DBSHFL* ⊥⊕δ	*	*	*
5	101	*	*	*	*	*	*	*	*
6	110	*	*	*	*	*	*	*	*
7	111	*	*	*	RDHWR ⊕	*	*	*	*

[1] Release 2 of the Architecture added the SPECIAL3 opcode. Implementations of Release 1 of the Architecture signaled a Reserved Instruction Exception for this opcode and all function field values shown above.

Table A.7 MIPS64 *MOVCI* Encoding of *tf* Bit

tf	*bit 16*	
	0	1
	MOVF	MOVT

Table A.8 MIPS64[1] *SRL* Encoding of Shift/Rotate

R	*bit 21*	
	0	1
	SRL	ROTR

[1] Release 2 of the Architecture added the ROTR instruction. Implementations of Release 1 of the Architecture ignored bit 21 and treated the instruction as an SRL

Table A.9 MIPS64[1] *SRLV* Encoding of Shift/Rotate

R	*bit 6*	
	0	1
	SRLV	ROTRV

[1] Release 2 of the Architecture added the ROTRV instruction. Implementations of Release 1 of the Architecture ignored bit 6 and treated the instruction as an SRLV

Table A.10 MIPS64[1] *DSRLV* Encoding of Shift/Rotate

R	*bit 6*	
	0	1
	DSRLV	DROTRV

[1] Release 2 of the Architecture added the DROTRV instruction. Implementations of Release 1 of the Architecture ignored bit 6 and treated the instruction as a DSRLV

Table A.11 MIPS64[1] *DSRL* Encoding of Shift/Rotate

R	*bit 21*	
	0	1
	DSRL	DROTR

[1] Release 2 of the Architecture added the DROTR instruction. Implementations of Release 1 of the Architecture ignored bit 21 and treated the instruction as a DSRL

Table A.12 MIPS64[1] *DSRL32* Encoding of Shift/Rotate

R	*bit 21*	
	0	1
	DSRL32	DROTR32

[1] Release 2 of the Architecture added the DROTR32 instruction. Implementations of Release 1 of the Architecture ignored bit 21 and treated the instruction as a DSRL32

Table A.13 MIPS64 *BSHFL* and *DBSHFL* Encoding of *sa* Field[1]

sa		*bits 8..6*							
		0	1	2	3	4	5	6	7
bits 10..9		000	001	010	011	100	101	110	111
0	00			WSBH (BSHFL) DSBH (DBSHFL)			DSHD (DBSHFL)		
1	01								
2	10	SEB (BSHFL)							
3	11	SEH (BSHFL)							

1. The *sa* field is sparsely decoded to identify the final instructions. Entries in this table with no mnemonic are reserved for future use by MIPS Technologies and may or may not cause a Reserved Instruction exception.

Table A.14 MIPS64 *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	DMFC0 ⊥	∗	∗	MTC0	DMTC0 ⊥	∗	∗
1	01	∗	∗	RDPGPR ⊕	*MFMC0*[1] δ⊕	∗	∗	WRPGPR ⊕	∗
2	10	C0 δ							
3	11								

1. Release 2 of the Architecture added the MFMC0 function, which is further decoded as the DI (bit 5 = 0) and EI (bit 5 = 1) instructions.

Table A.15 MIPS64 *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.16 MIPS64 *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	DMFC1 ⊥	CFC1	MFHC1 ⊕	MTC1	DMTC1 ⊥	CTC1	MTHC1 ⊕
1	01	*BC1* δ	*BC1ANY2* δε∇	*BC1ANY4* δε∇	∗	∗	∗	∗	∗
2	10	S δ	D δ	∗	∗	W δ	L δ	PS δ	∗
3	11	∗	∗	*	∗	*	∗	*	∗

Table A.17 MIPS64 *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	ROUND.L ∇	TRUNC.L ∇	CEIL.L ∇	FLOOR.L ∇	ROUND.W	TRUNC.W	CEIL.W	FLOOR.W
2	010	∗	MOVCF δ	MOVZ	MOVN	∗	RECIP ∇	RSQRT ∇	∗
3	011	∗	∗	∗	∗	RECIP2 ε∇	RECIP1 ε∇	RSQRT1 ε∇	RSQRT2 ε∇
4	100	∗	CVT.D	∗	∗	CVT.W	CVT.L ∇	CVT.PS ∇	∗
5	101	∗	∗	∗	∗	∗	∗	∗	∗
6	110	C.F CABS.F ε∇	C.UN CABS.UN ε∇	C.EQ CABS.EQ ε∇	C.UEQ CABS.UEQ ε∇	C.OLT CABS.OLT ε∇	C.ULT CABS.ULT ε∇	C.OLE CABS.OLE ε∇	C.ULE CABS.ULE ε∇
7	111	C.SF CABS.SF ε∇	C.NGLE CABS.NGLE ε∇	C.SEQ CABS.SEQ ε∇	C.NGL CABS.NGL ε∇	C.LT CABS.LT ε∇	C.NGE CABS.NGE ε∇	C.LE CABS.LE ε∇	C.NGT CABS.NGT ε∇

Table A.18 MIPS64 *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	ROUND.L ∇	TRUNC.L ∇	CEIL.L ∇	FLOOR.L ∇	ROUND.W	TRUNC.W	CEIL.W	FLOOR.W
2	010	∗	MOVCF δ	MOVZ	MOVN	∗	RECIP ∇	RSQRT ∇	∗
3	011	∗	*	∗	∗	RECIP2 ε∇	RECIP1 ε∇	RSQRT1 ε∇	RSQRT2 ε∇
4	100	CVT.S	∗	∗	∗	CVT.W	CVT.L ∇	∗	∗
5	101	∗	∗	∗	∗	∗	*	∗	∗
6	110	C.F CABS.F ε∇	C.UN CABS.UN ε∇	C.EQ CABS.EQ ε∇	C.UEQ CABS.UEQ ε∇	C.OLT CABS.OLT ε∇	C.ULT CABS.ULT ε∇	C.OLE CABS.OLE ε∇	C.ULE CABS.ULE ε∇
7	111	C.SF CABS.SF ε∇	C.NGLE CABS.NGLE ε∇	C.SEQ CABS.SEQ ε∇	C.NGL CABS.NGL ε∇	C.LT CABS.LT ε∇	C.NGE CABS.NGE ε∇	C.LE CABS.LE ε∇	C.NGT CABS.NGT ε∇

Table A.19 MIPS64 *COP1* Encoding of Function Field When *rs*=*W* or *L*[1]

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	∗	∗	∗	∗	CVT.PS.PW ε∇	∗
5	101	∗	∗	∗	∗	∗	∗	∗	∗
6	110	∗	∗	∗	∗	∗	∗	∗	∗
7	111	∗	∗	∗	∗	∗	∗	∗	∗

[1] Format type *L* is legal only if 64-bit floating point operations are enabled.

Table A.20 MIPS64 *COP1* Encoding of Function Field When *rs*=*PS*[1]

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 ∇	*	*	ABS ∇	MOV ∇	NEG ∇
1	001	*	*	*	*	*	*	*	*
2	010	*	MOVCF δ∇	MOVZ ∇	MOVN ∇	*	*	*	*
3	011	ADDR ε∇	*	MULR ε∇	*	RECIP2 ε∇	RECIP1 ε∇	RSQRT1 ε∇	RSQRT2 ε∇
4	100	CVT.S.PU ∇	*	*	*	CVT.PW.PS ε∇	*	*	*
5	101	CVT.S.PL ∇	*	*	*	PLL.PS ∇	PLU.PS ∇	PUL.PS ∇	PUU.PS ∇
6	110	C.F ∇ CABS.F ε∇	C.UN ∇ CABS.UN ε∇	C.EQ ∇ CABS.EQ ε∇	C.UEQ ∇ CABS.UEQ ε∇	C.OLT ∇ CABS.OLT ε∇	C.ULT ∇ CABS.ULT ε∇	C.OLE ∇ CABS.OLE ε∇	C.ULE ∇ CABS.ULE ε∇
7	111	C.SF ∇ CABS.SF ε∇	C.NGLE ∇ CABS.NGLEε∇	C.SEQ ∇ CABS.SEQ ε∇	C.NGL ∇ CABS.NGL ε∇	C.LT ∇ CABS.LT ε∇	C.NGE ∇ CABS.NGE ε∇	C.LE ∇ CABS.LE ε∇	C.NGT ∇ CABS.NGT ε∇

[1] Format type *PS* is legal only if 64-bit floating point operations are enabled.

Table A.21 MIPS64 *COP1* Encoding of *tf* Bit When *rs*=*S*, *D*, or *PS*, Function=*MOVCF*

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

Table A.22 MIPS64 *COP2* 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	MFC2 θ	DMFC2 θ⊥	CFC2 θ	MFHC2 θ⊕	MTC2 θ	DMTC2 θ⊥	CTC2 θ	MTHC2 θ⊕
1	01	BC2 θ	*	*	*	*	*	*	*
2	10	*C2* θδ							
3	11								

Table A.23 MIPS64 *COP1X* Encoding of Function Field[1]

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	LWXC1 ∇	LDXC1 ∇	*	*	*	LUXC1 ∇	*	*
1	001	SWXC1 ∇	SDXC1 ∇	*	*	*	SUXC1 ∇	*	PREFX ∇
2	010	*	*	*	*	*	*	*	*
3	011	*	*	*	*	*	*	ALNV.PS ∇	*
4	100	MADD.S ∇	MADD.D ∇	*	*	*	*	MADD.PS ∇	*
5	101	MSUB.S ∇	MSUB.D ∇	*	*	*	*	MSUB.PS ∇	*
6	110	NMADD.S ∇	NMADD.D ∇	*	*	*	*	NMADD.PS ∇	*
7	111	NMSUB.S ∇	NMSUB.D ∇	*	*	*	*	NMSUB.PS ∇	*

[1] COP1X instructions are legal only if 64-bit floating point operations are enabled.

A.3 Floating Point Unit Instruction Format Encodings

Instruction format encodings for the floating point unit are presented in this section. This information is a tabular presentation of the encodings described in tables Table A.16 and Table A.23 above.

Table A.24 Floating Point Unit Instruction Format Encodings

fmt field (bits 25..21 of COP1 opcode)		*fmt3* field (bits 2..0 of COP1X opcode)					
Decimal	**Hex**	**Decimal**	**Hex**	**Mnemonic**	**Name**	**Bit Width**	**Data Type**
0..15	00..0F	—	—	Used to encode Coprocessor 1 interface instructions (MFC1, CTC1, etc.). Not used for format encoding.			
16	10	0	0	S	Single	32	Floating Point
17	11	1	1	D	Double	64	Floating Point
18..19	12..13	2..3	2..3	Reserved for future use by the architecture.			
20	14	4	4	W	Word	32	Fixed Point
21	15	5	5	L	Long	64	Fixed Point
22	16	6	6	PS	Paired Single	2 × 32	Floating Point
23	17	7	7	Reserved for future use by the architecture.			
24..31	18..1F	—	—	Reserved for future use by the architecture. Not available for *fmt3* encoding.			

Appendix B

Revision History

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

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

Revision	Date	Description
0.95	March 12, 2001	External review copy of reorganized and updated architecture documentation.
1.00	August 29, 2002	Update based on all feedback received: • Fix bit numbering in FEXR diagram • Clarify the description of the width of FPRs in 32-bit implementations • Correct tag on FIR diagram. • Update the compatibility and subsetting rules to capture the current requirements. • Remove the requirement that a licensee must consult with MIPS Technologies when assigning SPECIAL2 function fields.
1.90	September 1, 2002	Update the specification with the changes due to Release 2 of the Architecture. Changes included in this revision are: • The Coprocessor 1 FIR register was updated with new fields and interpretations. • Update architecture and ASE summaries with the new instructions and information introduced by Release 2 of the Architecture.
2.00	June 8, 2003	Continue the update of the specification for Release 2 of the Architecture. Changes included in this revision are: • Correct the revision history year for Revision 1.00 (above). It should be 2002, not 2001. • Remove NOR, OR, and XOR from the 2-operand ALU instruction table.
2.50	July 1, 2005	Changes in this revision: • Correct the wording of the hidden modes section (see Section 2.2, "Compliance and Subsetting"). • Update all files to FrameMaker 7.1. • Allow shadow sets to be implemented without vectored interrupts or support for an external interrupt controller. In such an implementation, they are software-managed.
2.60	June 25, 2008	• COP3 no longer extendable by customer. • Section on Instruction fetches added - 1. fetches & endian-ness 2. fetches & CCA 3. self-modified code
2.61	December 5, 2009	• Fixed paragraph numbering between chapters. • FPU chapter didn’t make it clear that MADD/MSUB were non-fused.

Revision	Date	Description
3.00	March 25, 2010	• Changes for microMIPS. • List changes in Release 2.5+ and non-microMIPS changes in Release 3. • List PRA implementation options.
3.01	December 10, 2010	• Change Security Classification for microMIPS AFP versions.
3.02	March 06, 2010	• There is no persietent interpretation of FPR values between instructions. The interpretation comes from the instruction being executed. • Clarification that the PS format availability is solely defined by the FIR.PS bit.