MIPS® Architecture For Programmers Volume I-A: Introduction to the MIPS32® Architecture

Document Number: MD00082
Revision 6.01
August 20, 2014

Contents

Figures

Tables

Chapter 1

About This Book

The MIPS® Architecture For Programmers Volume I-A: Introduction to the MIPS32® 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 MIPS32® Architecture
- Volume I-B describes conventions used throughout the document set, and provides an introduction to the microMIPS32™ Architecture
- Volume II-A provides detailed descriptions of each instruction in the MIPS32® instruction set
- Volume II-B provides detailed descriptions of each instruction in the microMIPS32™ instruction set
- Volume III describes the MIPS32® and microMIPS32™ 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 MIPS32® Architecture. Beginning with Release 3 of the Architecture, microMIPS is the preferred solution for smaller code size. Release 6 removes MIPS16e: MIPS16e cannot be implemented with Release 6.
- Volume IV-b describes the MDMX™ Application-Specific Extension to the MIPS64® Architecture and microMIPS64™. It is not applicable to the MIPS32® document set nor the microMIPS32™ document set. With Release 5 of the Architecture, MDMX is deprecated. MDMX and MSA can not be implemented at the same time. Release 6 removes MDMX: MDMX cannot be implemented with Release 6.
- Volume IV-c describes the MIPS-3D® Application-Specific Extension to the MIPS® Architecture. Release 6 removes MIPS-3D: MIPS-3D cannot be implemented with Release 6.
- Volume IV-d describes the SmartMIPS®Application-Specific Extension to the MIPS32® Architecture and the microMIPS32™ Architecture . Release 6 removes SmartMIPS: SmartMIPS cannot be implemented with Release 6.
- Volume IV-e describes the MIPS® DSP Module to the MIPS® Architecture.
- Volume IV-f describes the MIPS® MT Module to the MIPS® Architecture.
- Volume IV-h describes the MIPS® MCU Application-Specific Extension to the MIPS® Architecture.
- Volume IV-i describes the MIPS® Virtualization Module to the MIPS® Architecture.
- Volume IV-j describes the MIPS® SIMD Architecture Module to the MIPS® Architecture.

1.1 Typographical Conventions

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

1.1.1 Italic Text

- is used for *emphasis.*
- is used for *bits*, *fields*, and *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* and *D.*
- 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.

1.1.4 Colored Text

RegisterGreen color and *italic* font are used for CP0 registers and CP0 register bits and fields. *RegisterGreen* color and*i* $italic_{subscript}$ fonts are used for CP0 register bits and fields when appended to the register name.

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.
- **UNDEFINED** behavior in privileged modes such as Kernel mode becomes UNPREDICTABLE behavior when virtualized and executed in Guest Kernel mode, as described in Volume IV-i, the *MIPS® Virtualization Module to the MIPS® Architecture*.

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 using a high-level language pseudocode resembling Pascal. Special symbols used in the pseudocode notation are listed in Table 1.1.

Table 1.1 Symbols Used in Instruction Operation Statements

Symbol	Meaning
←	Assignment
=, ≠	Tests for equality and inequality
||	Bit string concatenation
x^y	A *y*-bit string formed by *y* copies of the single-bit value *x*
b#n	A constant value *n* in base *b*. For instance 10#100 represents the decimal value 100, 2#100 represents the binary value 100 (decimal 4), and 16#100 represents the hexadecimal value 100 (decimal 256). If the "b#" prefix is omitted, the default base is 10.
0bn	A constant value *n* in base *2*. For instance 0b100 represents the binary value 100 (decimal 4).
0xn	A constant value *n* in base *16*. For instance 0x100 represents the hexadecimal value 100 (decimal 256).
$x_{y..z}$	Selection of bits *y* through *z* of bit string *x*. Little-endian bit notation (rightmost bit is 0) is used. If *y* is less than *z*, this expression is an empty (zero length) bit string.
x.bit[y]	Bit *y* of bitstring *x*. Alternative to the traditional MIPS notation x_y.
x.bits[y..z]	Selection of bits *y* through *z* of bit string *x*. Alternative to the traditional MIPS notation $x_{y..z}$.
x.byte[y]	Byte *y* of bitstring *x*. Equivalent to the traditional MIPS notation $x_{8*y+7..8*y}$.
x.bytes[y..z]	Selection of bytes *y* through *z* of bit string *x*. Alternative to the traditional MIPS notation $x_{8*y+7..8*z}$.
x.halfword[y] x.word[i] x.doubleword[i]	Similar extraction of particular bitfields (used in e.g., MSA packed SIMD vectors).
x.bit31, x.byte0, etc.	Examples of abbreviated form of x.bit[y], etc. notation, when y is a constant.
x.fieldy	Selection of a named subfield of bitstring *x*, typically a register or instruction encoding. More formally described as “Field y of register x”. For example, FIR_D = “the D bit of the Coprocessor 1 Floating-point Implementation Register (FIR)”.
+, −	2’s complement or floating point arithmetic: addition, subtraction
*, ×	2’s complement or floating point multiplication (both used for either)
div	2’s complement integer division
mod	2’s complement modulo
/	Floating point division
<	2’s complement less-than comparison
>	2’s complement greater-than comparison
≤	2’s complement less-than or equal comparison
≥	2’s complement greater-than or equal comparison
nor	Bitwise logical NOR
xor	Bitwise logical XOR
and	Bitwise logical AND
or	Bitwise logical OR

Table 1.1 Symbols Used in Instruction Operation Statements (Continued)

Symbol	Meaning
not	Bitwise inversion
&&	Logical (non-Bitwise) AND
<<	Logical Shift left (shift in zeros at right-hand-side)
>>	Logical Shift right (shift in zeros at left-hand-side)
GPRLEN	The length in bits (32 or 64) of the CPU general-purpose registers
GPR[x]	CPU general-purpose register *x*. The content of *GPR[0]* is always zero. In Release 2 of the Architecture, GPR[x] is a short-hand notation for SGPR[SRSCtlCSS, 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]*. Release 6 removes the floating point condition codes.
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)

Symbol	Meaning
I:, **I+n**:, **I-n**:	This occurs as a prefix to *Operation* description lines and functions as a label. It indicates the instruction time during which the pseudocode appears to "execute." Unless otherwise indicated, all effects of the current instruction appear to occur during the instruction time of the current instruction. No label is equivalent to a time label of **I**. Sometimes effects of an instruction appear to occur either earlier or later — that is, during the instruction time of another instruction. When this happens, the instruction operation is written in sections labeled with the instruction time, relative to the current instruction **I**, in which the effect of that pseudocode appears to occur. For example, an instruction may have a result that is not available until after the next instruction. Such an instruction has the portion of the instruction operation description that writes the result register in a section labeled **I+1**. The effect of pseudocode statements for the current instruction labeled **I+1** appears to occur "at the same time" as the effect of pseudocode statements labeled **I** for the following instruction. Within one pseudocode sequence, the effects of the statements take place in order. However, between sequences of statements for different instructions that occur "at the same time," there is no defined order. Programs must not depend on a particular order of evaluation between such sections.
PC	The *Program Counter* value. During the instruction time of an instruction, this is the address of the instruction word. The address of the instruction that occurs during the next instruction time is determined by assigning a value to *PC* during an instruction time. If no value is assigned to *PC* during an instruction time by any pseudocode statement, it is automatically incremented by either 2 (in the case of a 16-bit MIPS16e instruction) or 4 before the next instruction time. A taken branch assigns the target address to the *PC* during the instruction time of the instruction in the branch delay slot. In the MIPS Architecture, the PC value is only visible indirectly, such as when the processor stores the restart address into a GPR on a jump-and-link or branch-and-link instruction, or into a Coprocessor 0 register on an exception. Release 6 adds PC-relative address computation and load instructions. The PC value contains a full 32-bit address, all of which are significant during a memory reference.
ISA Mode	In processors that implement the MIPS16e Application Specific Extension or the microMIPS base architectures, the *ISA Mode* is a single-bit register that determines in which mode the processor is executing, as follows: **Encoding** / **Meaning** 0 — The processor is executing 32-bit MIPS instructions 1 — The processor is executing MIIPS16e or microMIPS instructions In the MIPS Architecture, the *ISA Mode* value is only visible indirectly, such as when the processor stores a combined value of the upper bits of PC and the *ISA Mode* into a GPR on a jump-and-link or branch-and-link instruction, or into a Coprocessor 0 register on an exception.
PABITS	The number of physical address bits implemented is represented by the symbol PABITS. As such, if 36 physical address bits were implemented, the size of the physical address space would be $2^{PABITS} = 2^{36}$ bytes.
FP32RegistersMode	Indicates whether the FPU has 32-bit or 64-bit floating point registers (FPRs). In MIPS32 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 Release 2 and Release 3) the FPU has 32 64-bit FPRs in which 64-bit data types are stored in any FPR. In MIPS32 Release 1 implementations, **FP32RegistersMode** is always a 0. MIPS64 implementations have a compatibility mode in which the processor references the FPRs as if it were a MIPS32 implementation. In such a case **FP32RegisterMode** is computed from the FR bit in the *Status* register. If this bit is a 0, the processor operates as if it had 32, 32-bit FPRs. If this bit is a 1, the processor operates with 32 64-bit FPRs. The value of **FP32RegistersMode** is computed from the FR bit in the *Status* register.

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 Notation for Register Field Accessibility

In this document, the read/write properties of register fields use the notations shown in Table 1.2.

Table 1.2 Read/Write Register Field Notation

Read/Write Notation	Hardware Interpretation	Software Interpretation
R/W	A field in which all bits are readable and writable by software and, potentially, by hardware. Hardware updates of this field are visible by software read. Software updates of this field are visible by hardware read. If the Reset State of this field is ‘‘Undefined’’, either software or hardware must initialize the value before the first read will return a predictable value. This should not be confused with the formal definition of **UNDEFINED** behavior.	
R	A field which is either static or is updated only by hardware. If the Reset State of this field is either ‘‘0’’, ‘‘Preset’’, or ‘‘Externally Set’’, hardware initializes this field to zero or to the appropriate state, respectively, on power-up. The term ‘‘Preset’’ is used to suggest that the processor establishes the appropriate state, whereas the term ‘‘Externally Set’’ is used to suggest that the state is established via an external source (e.g., personality pins or initialization bit stream). These terms are suggestions only, and are not intended to act as a requirement on the implementation. If the Reset State of this field is ‘‘Undefined’’, hardware updates this field only under those conditions specified in the description of the field.	A field to which the value written by software is ignored by hardware. Software may write any value to this field without affecting hardware behavior. Software reads of this field return the last value updated by hardware. If the Reset State of this field is ‘‘Undefined’’, software reads of this field result in an **UNPREDICTABLE** value except after a hardware update done under the conditions specified in the description of the field.

Table 1.2 Read/Write Register Field Notation (Continued)

Read/Write Notation	Hardware Interpretation	Software Interpretation
R0	Reserved, read as zero, ignore writes by software. Hardware ignores software writes to an R0 field. Neither the occurrence of such writes, nor the values written, affects hardware behavior. Hardware always returns 0 to software reads of R0 fields. The Reset State of an R0 field must always be 0. If software performs an mtc0 instruction which writes a non-zero value to an R0 field, the write to the R0 field will be ignored, but permitted writes to other fields in the register will not be affected.	**Architectural Compatibility:** R0 fields are reserved, and may be used for not-yet-defined purposes in future revisions of the architecture. When writing an R0 field, current software should only write either all 0s, or, preferably, write back the same value that was read from the field. Current software should not assume that the value read from R0 fields is zero, because this may not be true on future hardware. Future revisions of the architecture may redefine an R0 field, but must do so in such a way that software which is unaware of the new definition and either writes zeros or writes back the value it has read from the field will continue to work correctly. Writing back the same value that was read is guaranteed to have no unexpected effects on current or future hardware behavior. (Except for non-atomicity of such read-writes.) Writing zeros to an R0 field may not be preferred because in the future this may interfere with the operation of other software which has been updated for the new field definition.
0	**Release 6** Release 6 legacy “0” behaves like R0 - read as zero, nonzero writes ignored. Legacy “0” should not be defined for any new control register fields; R0 should be used instead.	
	HW returns 0 when read. HW ignores writes.	Only zero should be written, or, value read from register.
	pre-Release 6 pre-Release 6 legacy “0” - read as zero, nonzero writes are UNDEFINED.	
	A field which hardware does not update, and for which hardware can assume a zero value.	A field to which the value written by software must be zero. Software writes of non-zero values to this field may result in **UNDEFINED** behavior of the hardware. Software reads of this field return zero as long as all previous software writes are zero. If the Reset State of this field is ‘‘Undefined’’, software must write this field with zero before it is guaranteed to read as zero.

Table 1.2 Read/Write Register Field Notation (Continued)

Read/Write Notation	Hardware Interpretation	Software Interpretation
R/W0	Like R/W, except that writes of non-zero to a R/W0 field are ignored (e.g., write to $Status_{NMI}$).	
	Hardware may set or clear an R/W0 bit. Hardware ignores software writes of nonzero to an R/W0 field. Neither the occurrence of such writes, nor the values written, affects hardware behavior. Software writes of 0 to an R/W0 field may have an effect. Hardware may return 0 or nonzero to software reads of an R/W0 bit. If software performs an mtc0 instruction which writes a non-zero value to an R/W0 field, the write to the R/W0 field will be ignored, but permitted writes to other fields in the register will not be affected.	Software can only clear an R/W0 bit. Software writes 0 to an R/W0 field to clear the field. Software writes nonzero to an R/W0 bit in order to guarantee that the bit is not affected by the write.
W0	Like R/W0, except that the field cannot be read directly, but only through a level of indirection. An example is the UNFR COP1 register. Writes of non-zero to a W0 field are ignored.	
	Hardware may clear a W0 bit. Hardware ignores software writes of nonzero to a W0 field. Neither the occurrence of such writes, nor the values written, affects hardware behavior.	Software can only clear a W0 bit. Software writes 0 to a W0 field to clear the field. Software writes nonzero to an W0 bit in order to guarantee that the bit is not affected by the write.

1.5 For More Information

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

.
.

Chapter 2

Overview of the MIPS® Architecture

MIPS32® and MIPS64® architectures are high performance, industry-standard architectures that provide a robust and streamlined instruction set, with scalability from 32-bits to 64-bits, and are supported by a broad array of hardware and software development tools, including compilers, debuggers, in-circuit emulators, middleware, application platforms, and reference designs.

The MIPS architecture is based on a fixed-length, regularly encoded instruction set and uses a load/store data model, in which all operations are performed on operands in processor registers, and main memory is accessed only by load and store instructions. The load/store model reduces the number of memory accesses, thus easing memory bandwidth requirements, simplifies the instruction set, and makes it easier for compilers to optimize register allocation.

2.1 Historical Perspective

The MIPS Architecture has evolved over time from the original MIPS I™, through the MIPS V™, to the current MIPS32, MIPS64, and microMIPS™ architectures. Throughout the evolution of the architecture, each new ISA has been backward-compatible with previous ISAs. In the MIPS III™ ISA, 64-bit integers and addresses were added to the instruction set. The MIPS IV™ and MIPS V™ ISAs added improved floating-point operations and a new set of instructions that improved the efficiency of generated code and of data movement. Because of the strict backward-compatible requirement of ISAs, such changes were unavailable to 32-bit implementations of the ISA that were, by definition, MIPS I™ or MIPS II™ implementations. The MIPS32 Release 6 ISA maintains backward-compatibility, with the exception of a few rarely used instructions, though the use of trap-and-emulate or trap-and-patch; all pre-Release 6 binaries can execute under binary translation.

While the user-mode ISA was always backward-compatible, the PRA and the privileged-mode ISA were 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 optional components—Modules of

the base architecture, MIPS Application Specific Extensions (ASEs), User Defined Instructions (UDIs), and custom coprocessors to address the specific needs of particular markets.

The MIPS32 and MIPS64 Architectures provide 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 provide the same functionality as MIPS32 and MIPS64, with the additional benefit of smaller code size. The microMIPS architectures are supersets of MIPS32/MIPS64 architectures, with almost the same sets of 32-bit sized instructions and additional 16-bit instructions that reduce code size. Unlike the earlier versions of the architecture, microMIPS provides assembler-source code compatibility with its predecessors instead of binary compatibility.

2.2 Components of the MIPS® Architecture

2.2.1 MIPS Instruction Set Architecture (ISA)

The MIPS32 and MIPS64 Instruction Set Architectures define a compatible family of instructions that handle 32-bit data and 64-bit data (respectively) within the framework of the overall MIPS Architecture. 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 programs executing on any MIPS32 or MIPS64 processor; all instructions in the MIPS64 ISA are backward compatible with those instructions in the MIPS32 ISA. In many cases, privileged programs are also object-code compatible—using conditional compilation or assembly language macros, it is often possible to write privileged programs that run on both MIPS32 and MIPS64 implementations.

In Release 6 implementations, object-code compatibility is not guaranteed when directly executing pre-Release 6 code, because certain pre-Release 6 instruction encodings are allocated to completely different instructions on Release 6. Nevertheless, there is a useful subset of instructions that have the same encodings in both Release 6 and pre-Release 6, and an even larger subset that can be trapped and emulated. Furthermore, using conditional compilation or assembly language macros, it is often possible to write software that runs on both Release 6 and pre-Release 6 implementations. Binary compatibility can be obtained by binary translation; Release 6 is designed so that simple instruction replacement can accomplish all such binary translation, minimizing remapping of instruction addresses.

For example, to binary translate/patch a pre-Release 6 binary, the Release 6 compact branch instructions, with no delay slots, mean that any instruction can be replaced by a BALC single instruction call to an emulation function - assuming that the emulation function can be reached by the branch target with its 26 bit / 256MB span, and that the link register can be overwritten - which a binary translator can usually arrange. A single BC instruction avoids using the link register, at the cost of more emulation entry points. JC/JIALC can also be used, although their smaller 16-bit offset probably requires the binary translator to use an extra register.

Release 6 and subsequent releases will be backward compatible going forward, i.e., Release 6 code will run on all subsequent releases.

2.2.2 MIPS Privileged Resource Architecture (PRA)

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

2.2.3 MIPS Modules and Application Specific Extensions (ASEs)

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

2.2.4 MIPS User Defined Instructions (UDIs)

In addition to support for ASEs and Modules as described above, the MIPS32 and MIPS64 Architectures define specific instructions for use by each implementation. The *Special2* and/or *COP2* major opcodes and Coprocessor 2 are reserved for capabilities defined by each implementation. In Release 6, use of the *Special2* opcode is not permitted.

2.3 Evolution of the Architecture

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:

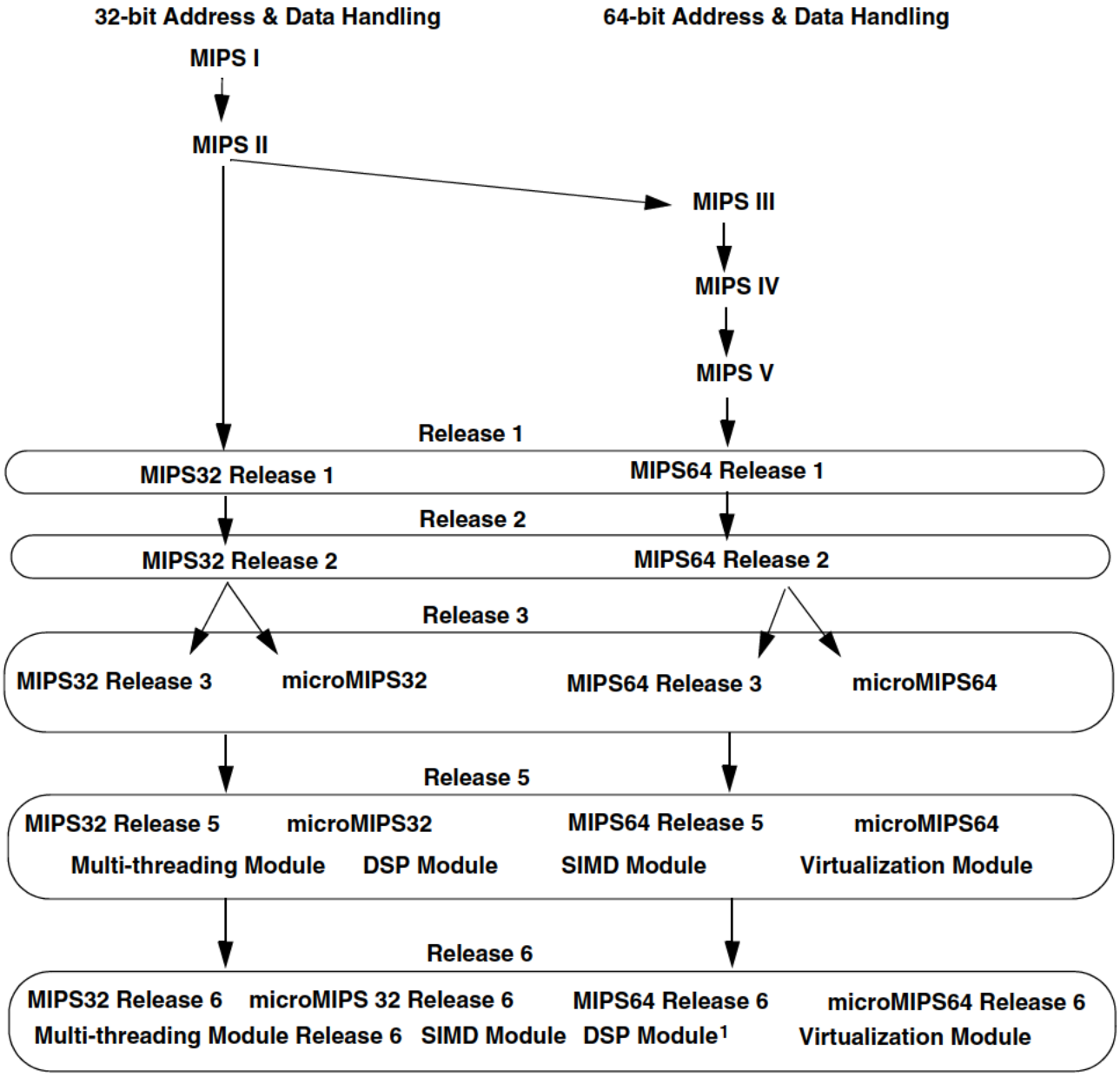
- are applicable to a wide market
- provide long-term benefit
- maintain architectural scalability
- are standardized to prevent fragmentation
- are a superset of the existing architecture

Taking into account these criteria, architects at MIPS Technologies constantly evaluate suggestions for architectural changes and enhancements, and new releases of the architecture, while infrequent, have been made at appropriate points:

- Release 1, the original version of the MIPS32 Architecture, released in 1985
- Release 2, added in 2002
- Release 3 (MIPSr3™), added in 2010
- Release 4, added in 2012. For internal use only.
- Release 5, added in 2013
- Release 6 added in 2014

The evolution of the MIPS architecture is summarized in Figure 2.1

Figure 2.1 MIPS Architecture Evolution



1. Use of MSA/SIMD Module is strongly encouraged in place of DSP. DSP Module and MSA/SIMD Module cannot be implemented together.

2.3.1 MIPS I through MIPS V Architectures

Changes to the MIPS I through MIPS V Architectures are described below.

- 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) that 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 in the MIPS32 Architecture, leaving the restriction that the effective address be naturally-aligned.

- Early MIPS implementations required two instructions separating a *MFLO* or *MFHI* from the next integer multiply or divide operation. This hazard was eliminated in the MIPS IV ISA, although the MIPS RISC Architecture Specification does not clearly explain this fact. The MIPS32 Architecture explicitly eliminates this hazard and requires that the hi and lo registers be fully interlocked in hardware for all integer multiply and divide instructions (including, but not limited to, the *MADD*, *MADDU*, *MSUB*, *MSUBU*, and *MUL* instructions introduced in this specification).

- The Implementation and Programming Notes included in the instruction descriptions for the *MADD*, *MADDU*, *MSUB*, *MSUBU*, and *MUL* instructions should be applied to all integer multiply and divide instructions in the MIPS RISC Architecture Specification.

2.3.2 MIPS32 Architecture Release 2

Enhancements in Release 2 of the MIPS32 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 Memory: These changes provide support for a 1KByte page size. This change is optional in Release 2 implementations, and support is denoted by $Config3_{SP}$.

2.3.3 MIPS32 Architecture Releases 2.5+

Some optional features were added after Revision 2.5:

- 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.

2.3.4 MIPS32 Release 3 Architecture (MIPSr3™)

MIPSr3™ is a family of architectures that includes Release 3.0 of the MIPS32 Architecture and the first release of the microMIPS32 architecture.

Enhancements in the MIPS Release 3 Architecture are:

- microMIPS instruction set.
  - This instruction set contains both 16-bit and 32-bit sized instructions.
  - The microMIPS32 ISA has all of the functionality of MIPS32 with smaller code sizes.
  - microMIPS32 is assembler source code compatible with MIPS32.

  - microMIPS32 replaces the MIPS16e ASE.
  - microMIPS32 is an additional base instruction set architecture that is supported along with MIPS32.
  - A device can implement either the base ISA or both. The ISA field of the *Config3* register denotes which ISA is implemented.
  - A device can implement any other Module/ASE with either base architecture.
  - microMIPS32 shares the same privileged resource architecture with MIPS32.
  - Branch Likely instructions are not supported in the microMIPS hardware architecture. 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 the *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 the *RXI* field of *Config3*.

- A more programmable virtual address space map without fixed cacheability and mapability attributes is introduced as an optional feature. This allows implementations to decide how large/small uncached/unmapped segments need to be. These capabilities are implemented through the Segmentation Control registers. This optional feature is denoted by the *SC* field of *Config3*.

- Along with a programmable virtual address map, it is possible to create separate user-mode & kernel-mode views of segments. This allows a larger kernel virtual address space to be defined. To access both this larger kernel address space and the overlapping user-space, additional load/store instructions are introduced. These new optional instructions are denoted by the *EVA* field of *Config5*.

- Support for certain IEEE-754-2008 FPU behaviors (as opposed to behaviors of the older IEEE-754-1985 standard) is now defined. These behaviors are indicated by the *Has2008* field of the *FIR* register in the FPU and bits *ABS2008* or *NAN2008* in the *FCSR* register.

- Optional TLB invalidate instructions are introduced. These are required for Segmentation Control that allows creation of a virtual address map without unmapped segments.

2.3.5 MIPS32 Architecture Release 5

Release 5 is a family of architectures (MIPS32, MIPS64, microMIPS32, and microMIPS64) that adds the following capabilities:

- The Multi-threading module is now an optional component of all of the base architectures. Previously the MT ASE was licensed as a separate architecture product.

- The DSP module is now an optional component of all the base architectures. Previously, the DSP ASE was licensed as a separate architecture product.

- The Virtualization module is now an optional component of all the base architectures.

- The MIPS SIMD Architecture (MSA) module is now an optional component of all the base architectures.

Release 5 has the following changes:

- The MDMX ASE is formally deprecated. The equivalent functionality is provided by the MSA module.
- The 64-bit versions of the DSP ASE are formally deprecated. The equivalent functionality is provided by the MSA module.
- If an FPU is present, it must be a 64-bit FPU.
- The MIPS32 and MIPS64 Release 5 architectures provide no features that support IEEE-754-2008 fused multiply-add without intermediate rounding. (In Release 6, unfused multiply-adds are removed, and fused multiply-adds are added.)

2.3.6 MIPS32 Architecture Release 6

Release 6 is a family of architectures (MIPS32 and MIPS64) that adds the following capabilities:

- The instruction set has been simplified by removing infrequently used instructions and rearranging instruction encodings so as to free a significant part of the opcode map for future expansion.
- CPU Enhancements
  - Some 3-source instructions (conditional moves) are removed.
  - Branch Likely instructions are removed (they were deprecated in earlier releases).
  - A powerful family of compact branches with no delay slot, including: unconditional branch (BC) and branch-and-link (BALC) with a very large 26-bit offset (+/- 128 MB span); conditional branch on zero/non-zero with a large 21-bit offset (+/- 4MB span); a full set of signed and unsigned conditional branches that compare two registers (e.g., BGTUC), or which compare a register against zero (e.g., BGTZC); and a full set of branch and link instructions that compare a register against zero (e.g., BGTZALC).
  - Integer overflow: Some trapping instructions are removed, specifically those with 16-bit immediate fields (e.g., ADDI (trap on integer overflow), TGEI (compare and trap)), mitigated by compact branches on overflow / no- overflow, which is easier to use by software.
  - Compact Indexed Jump instructions with no delay slot: Designed to support large absolute addresses.
  - Instructions to generate large constants, loading (adding) constants to bits 16-31, 32-47, and 48-63.
  - PC-relative instructions: In addition to branches and jumps, loads of 32- and 64-bit data and address generation with large relative offsets. Release 6 has true PC+offset relative-addressing control-transfer instructions that can span up to 26 bits (256MB), without the alignment restriction of the Jump (J) instruction (which can still be used in Release 6).
  - Integer accumulator instructions and the *HI*/*LO* registers are removed from the Release 6 base instruction set and moved to the DSP Module.
  - Bit-reversal and byte-alignment instructions migrated from DSP to Release 6 base instruction set.
  - Multiply and Divide instructions are redefined to produce a single GPR result.
  - The unaligned memory instructions are removed (e.g., LWL/LWR) and replaced by requiring misaligned memory access for most ordinary load/store instructions (possibly via trap-and-emulate).

- New instruction BALIGN can be used to emulate a misaligned load without using LWL/LWR, following a pair of ordinary load words.
- CPU truth values changed from single-bit to multi-bit: pre-Release 6 instructions that only looked at bit 0 of the register containing a truth value are replaced by Release 6 instructions that generate truth values of all zeroes or all ones (suitable for logical operations involving masks) and interpret all zeroes or any non-zero bit as true or false, which is compatible with programming languages such as C. There are also related changes to branches and conditional move instructions.
- Indexed addressing is removed for FPU loads and stores (e.g., LWXC1), mitigated by left shift add instructions (e.g., LSA rd:=rs<<scale+rt).
- Changes to 32-bit addressing in MIPS64: pre-Release 6 $Status_{UX}$ sign extension of addresses from bit 31 is only applied to user-mode data memory references. Release 6 extends this to instruction fetch and to privileged memory references (e.g., kernel).
- Instructions re-encoded to save opcode space: for example, Coprocessor 2, SPECIAL2, atomic (e.g., LL/SC), cache, and prefetch JR and JALR.
- Changes to SPECIAL2 instructions and UDIs: Release 6 reserves the SPECIAL2 instruction encodings; COP2 instructions remain available to customers.

- FPU Enhancements
  - The FR=0 FPU register model, in which 64-bit datatypes (D/L) are stored in even-odd pairs of 32-bit registers, is eliminated. The FPU must be 64-bit. If a 32-bit FPU is supported, $FIR_{D/L}$ must be zero.
  - Use of single-precision formats only is now permitted: Implementations with single-precision (S and W), but without double-precision (D and L), are allowed.
  - Added features that support trap-and-emulation of double-precision and MSA instructions, and hardware implementations with less than full 64-bit register widths.
  - FPU and MSA instructions that use 64-bit and 128-bit registers, respectively, overlaid on 32-bit registers leave the upper bits UNPREDICTABLE. This behavior facilitates trap-and-emulate.
  - Floating-point condition codes are removed; New instructions (CMP.condn.fmt) generate masks of all 0s and all 1s that are stored in FPRs. New instructions test bit 0, the least-significant bit, of an FPR, for both branches (BC1EQZ/BC1NEZ) and branchless selections (SEL.fmt, SELEQZ.fmt, SELNEZ.fmt). Old instructions that use the FCCs are removed.
  - IEEE 2008: NaN behavior is required. IEEE 2008 instruction-set support, such as minimum and maximum, is required. $FCSR_{Has2008}$=$FIR_{ABS2008}$=$FIR_{NAN2008}$=1 are hardwired read-only. Fused multiply-add instructions (MADDF.fmt, MSUBF.fmt) are required; non-fused multiply-add instructions that were available in previous versions of the MIPS ISA are removed.
  - Paired single (PS) and MIPS-3D are not allowed.
  - Indexed addressing removed: Modes adding two registers are removed, mitigated by instructions that generate such addresses.

2.4 Compliance and Subsetting

To be compliant with the MIPS32 Architecture, designs must implement a set of required features, as described in this document set. To allow flexibility in implementations, the MIPS32 Architecture provides subsetting rules. An implementation that follows these rules is compliant with the MIPS32 Architecture as long as it adheres strictly to the rules and fully implements the remaining instructions.

Supersetting of the MIPS32 Architecture is only allowed by adding functions to the *SPECIAL2* and/or *COP2* major opcodes, by adding control for co-processors via the *COP2*, *LWC2*, *SWC2*, *LDC2*, and/or *SDC2*, or by the addition of approved Application Specific Extensions or Modules.

However, Release 6 removes all instructions under the SPECIAL2 major opcode, either by removing them or moving them to the COP2 major opcode. Similarly, all Coprocessor 2 support instructions (e.g., LWC2) have been moved to the COP2 major opcode. Supersetting of the Release 6 architecture is only allowed in the COP2 major opcode, or by the addition of approved Application Specific Extensions or Modules. SPECIAL2 is reserved for MIPS.

Note: The use of COP3 as a customizable coprocessor has been removed in Release 2 of the MIPS32 architecture. The use of the COP3 is now reserved for future extensions of the architecture. Implementations using Release1 of the MIPS32 architecture are strongly discouraged from using the COP3 opcode for a user-available coprocessor as doing so will limit the potential for an upgrade path to a 64-bit floating point unit.

New features provided by a release of the architecture, both optional and required, must be consistent within a given implementation. When a new feature is implemented, all other features must be implemented in a manner consistent with that release.

2.4.1 Subsetting of Non-Privileged Architecture

- All non-privileged CPU instructions must be implemented. No subsetting of these instructions is permitted, as per the MIPS Instruction Set Architecture release supported.
- For any instruction that is subsetted out, in compliance with the rules below, an attempt to execute that instruction must cause the appropriate exception (typically Reserved Instruction or Coprocessor Unusable).
- The FPU and related support instructions, such as CPU conditional branches on FPU conditions (e.g., pre-Release 6 BC1T/BC1F, Release 6 BC1NEQZ) and CPU conditional moves on FPU conditions (e.g., pre-Release 6 MOVT/MOVF), may be omitted. Software can determine if an FPU is implemented by checking the state of the *FP* bit in the *Config1* CP0 register. Software can determine which FPU data types are implemented by checking the appropriate bits in the *FIR* CP1 register. The following allowable FPU subsets are compliant with the MIPS32 architecture:
  - No FPU

    $Config1_{FP}$=0
  - FPU with S and W formats and all supporting instructions

    This 32-bit subset is permitted in Release 6, but prohibited in releases prior to Release 6.

    $Config1_{FP}$=1, $Status_{FR}$=0, FIR_S=FIR_L=1, FIR_D=FIR_L=FIR_{PS}=0

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

  $Config1_{FP}$=1, $Status_{FR}$=(see below), FIR_S=FIR_L=FIR_D=FIR_L=1, FIR_{PS}=0

  pre-Release 5 permits this 64-bit configuration and allows both FPU register modes. $Status_{FR}$=0 support is required but $Status_{FR}$=1 support is optional.

  Release 5 permits this 64-bit configuration and requires both FPU register modes, i.e., both $Status_{FR}$=0 and $Status_{FR}$=1 support are required.

  Release 6 permits this 64-bit configuration but requires $Status_{FR}$=1 and FIR_{F64}=1. Release 6 prohibits $Status_{FR}$=0 if FIR_D=1 or FIR_L=1.

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

  $Config1_{FP}$=1, $Status_{FR}$=0/1, FIR_S=FIR_L=FIR_D=FIR_L=FIR_{PS}=1

  Release 6 prohibits this mode, and any mode with FIR_{PS}=1 paired-single support.

- Coprocessor 2 is optional and may be omitted. Software can 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, LDC2, LWC2, MFC2, MTC2, SDC2, and SWC2) may be omitted on an instruction-by-instruction basis.

- The caches are optional. The $Config1_{DL}$ and $Config1_{IL}$ fields denote whether the first-level caches are present or not.

- 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 Modules and ASEs are optional and may be subsetted out. If most cases, software can determine if a supported Module or ASE is implemented by checking the appropriate bit in the *Config1*, *Config3*, or *Config4* CP0 register. If they are implemented, they must implement the entire ISA applicable to the component, or implement subsets that are approved by the Module and 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 EJTAG is not implemented, the EJTAG instructions (SDBBP and DERET) can be subsetted out.

- In 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), and for this reason, if a device implements both architecture branches, the address and data widths must be consistent. If a device implements MIPS64 and also implements microMIPS, it must implement microMIPS64, not just microMIPS32. Similarly, if a device implements microMIPS64 and also implements MIPS32 or MIPS64, it must implement MIPS64, not just MIPS32.

- The JALX instruction is required if and only if ISA mode-switching is supported. If both 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 is implemented, and MIPS16e is not implemented, then the JALX instruction is not implemented. Release 6 removes the JALX instruction.

2.4.2 Subsetting of Privileged Architecture

Some of the non-privileged subsetting rules described in the preceding section also apply to the privileged architecture:

- Coprocessor 2 is optional.
- Caches are optional.
- Reserved and "0" bit indications in instructions and control register fields.
    - EJTAG is optional.
    - Multiple instruction set support: architecture branches (MIPS32/64 and microMIPS32/64), MIPS16e, register widths and the JALX instruction.
    - Co-dependence of architecture features.

Subsetting rules for features less visible to the non-privileged code include:

- 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., the 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 can determine the type of the MMU by checking the *MT* field in the *Config* CP0 register.
- The EVA load/store instructions (LWE, LHE, LBE, LBUE, LHUE, SWE, SHE, SBE) are optional.
- 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 Privileged Resource Architecture includes several implementation options and may be subsetted in accordance with those options. Some of those options are:
    - Interrupt Modes
    - Shadow Register Sets
    - Common Device Memory Map
    - Parity/ECC support
    - *UserLocal* register (required in Release 6)
    - *ContextConfig* register
    - *PageGrain* register
    - *Config1-4* registers
    - *Performance Counter*, *WatchPoint*, and *Trace* Registers

- Cache control/diagnostic registers
- Kernel-mode scratch registers (required in Release 6)

Chapter 3

Modules and Application Specific Extensions

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

3.1 Description of Optional Components

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 components are provided for use with the base ISAs (MIPS32/MIPS64 and microMIPS32/microMIPS64MIPS32 and MIPS64).

These optional components are licensed to MIPS architecture licensees in two different ways:

1. Modules - these optional components are part of the Base Architecture (Revision 5 and newer). If a company has licensed one of the base architectures from MIPS Technologies, then that company has also rights to implement any of the associated modules of that base architecture.

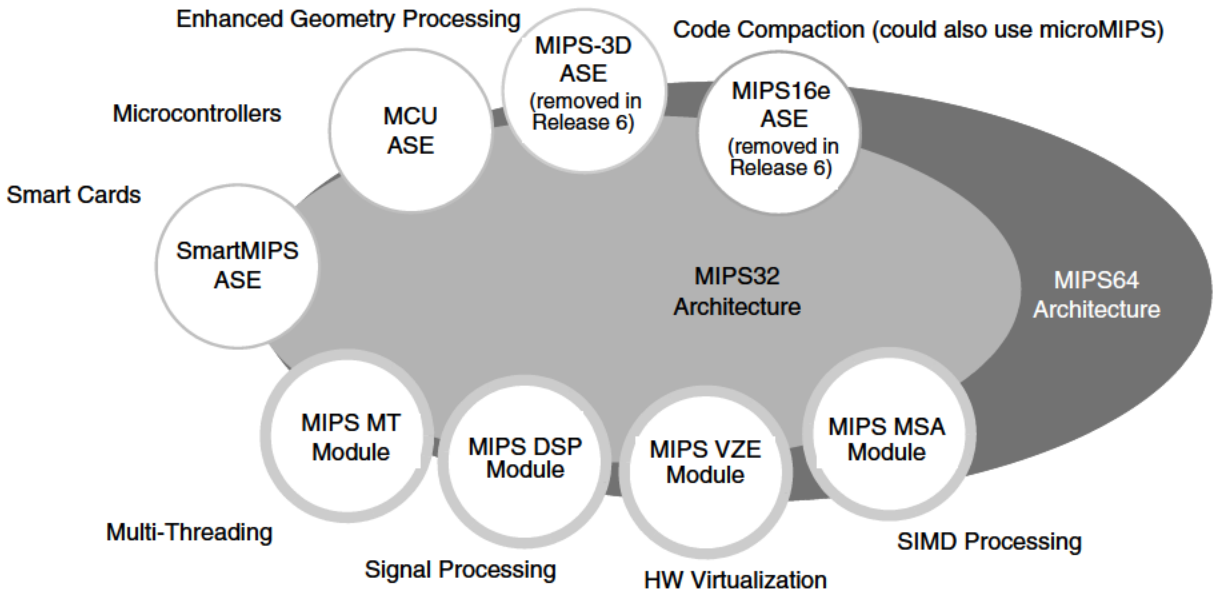
2. Application Specific Extensions - these optional components are sold as separate architecture products from MIPS Technologies.

The Modules and 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. A Module or ASE can be used with the appropriate ISA to meet the needs of a specific application or an entire class of applications.

Figure 3.1 shows the relationship of the ASEs to the MIPS32 and MIPS64 ISAs.

Figure 3.1 MIPS ISAs, ASEs, and Modules



The MIPS32 Architecture is a strict subset of the MIPS64 Architecture. MIPS32 Release 6 is a strict subset of MIPS64 Release 6. 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 Application Specific Instructions

As of the published date of this document, the following Application Specific Extensions are supported by the architecture.

Component	Module or ASE	Supported Base Architectures	Use
MIPS16e™	ASE	MIPS32 and MIPS64[1]	Code Compaction
MIPS-3D®	ASE	MIPS32 and MIPS64[1]	Geometry Processing
SmartMIPS®	ASE	MIPS32	Smart Cards and Smart Objects
MIPS® DSP	Module	MIPS32 and MIPS64[2]	Signal Processing
MIPS® MT	Module	MIPS32 and MIPS64	Multi-Threading
MCU	ASE	MIPS32 and MIPS64	Fast Interrupt Response and I/O register programming
VZE	Module	MIPS32 and MIPS64	Hardware Support for Virtualization
MSA	Module	MIPS32 and MIPS64	SIMD support

1. Not supported in Release 6.
2. MSA is recommended substitute. DSP is not allowed if MSA is implemented. Use of DSP Module is strongly discouraged from Release 6 onwards.

3.2.1 MIPS16e™ Application Specific Extension

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.

The MIPS16e ASE is not allowed with Release 6.

3.2.2 MDMX™ Application Specific Extension

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.The MDMX ASE is not supported in Release 6; the same functionality is provided by the SIMD Module. Because the MDMX ASE requires a 64-bit Architecture, it is not discussed in this document set.Volume IV-b of this document set describes the MDMX ASE.

3.2.3 MIPS-3D® Application Specific Extension

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. The MIPS-3D ASE is not available with Release 6.

3.2.4 SmartMIPS® Application Specific Extension

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. Volume IV-d of this document set describes the SmartMIPS ASE. Because the SmartMIPS ASE requires the MIPS32 Architecture, it is not discussed in the MIPS64 document set. SmartMIPS, however, is not supported in Release 6.

3.2.5 MIPS® DSP Module

The MIPS DSP Module 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 Module.

3.2.6 MIPS® MT Module

The MIPS MT Module 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 Module. Release 6 Multi-threading specification supersedes Volume IV-f.

3.2.7 MIPS® MCU Application Specific Extension

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.

3.2.8 MIPS® Virtualization Module

The MIPS Virtualization Module provides hardware acceleration of virtualization of Operating Systems. Volume IV-i of this document set describes the MIPS VZ Module.

3.2.9 MIPS® SIMD Architecture Module

The MIPS SIMD Architecture Module provides high performance parallel processing of vector operations through the use of 128-bit wide vector registers. Volume IV-j of this document set describes the MIPS MSA Module.

Chapter 4

CPU Programming Model

This chapter describes the following aspects of the MIPS32 CPU programming model:

- CPU Data Formats
- Coprocessors (CP0-CP3)
- CPU Registers
- Byte Ordering and Endianness
- Memory Access Types
- Implementation-Specific Access Types
- Cacheability and Coherency Attributes and Access Types
- Mixing Access Types
- Instruction Fetch

Other aspects of the Programming Model such as modes of operation, virtual memory, and the handling of interrupts and exceptions, are described in Volume III. The instruction set is described in Chapter 5, "CPU Instruction Set" on page 52 and in detail in Volume II.

4.1 CPU Data Formats

The CPU defines the following data formats:

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

4.2 Coprocessors (CP0-CP3)

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

1. The CPU doubleword and FPU floating point paired-single and Long fixed-point data formats are available in Release 1 implementations of the MIPS64 Architecture and in Release 2 (or subsequent releases) implementations that include a 64-bit floating point unit, whether MIPS32, MIPS64, microMIPS32, or microMIPS64.

- 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 Release 1 implementations of the MIPS64 Architecture and in 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 modes. CP0 also controls the cache subsystem and provides diagnostic control and error recovery facilities. The architectural features of CP0 are defined in Volume III.

4.3 CPU Registers

The MIPS32 Architecture defines the following CPU registers:

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

4.3.1 CPU General-Purpose Registers

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

- *r0* is hard-wired to a value of zero. It can be used as the target register for any instruction whose result is to be discarded and can be used as a source when a zero value is needed.
- *r31* is the destination register used by procedure call branch/jump and link instructions (e.g., JAL and Release 6 JALC) 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.

4.3.2 CPU Special-Purpose Registers

Prior to Release 6, the CPU contained three special-purpose registers.

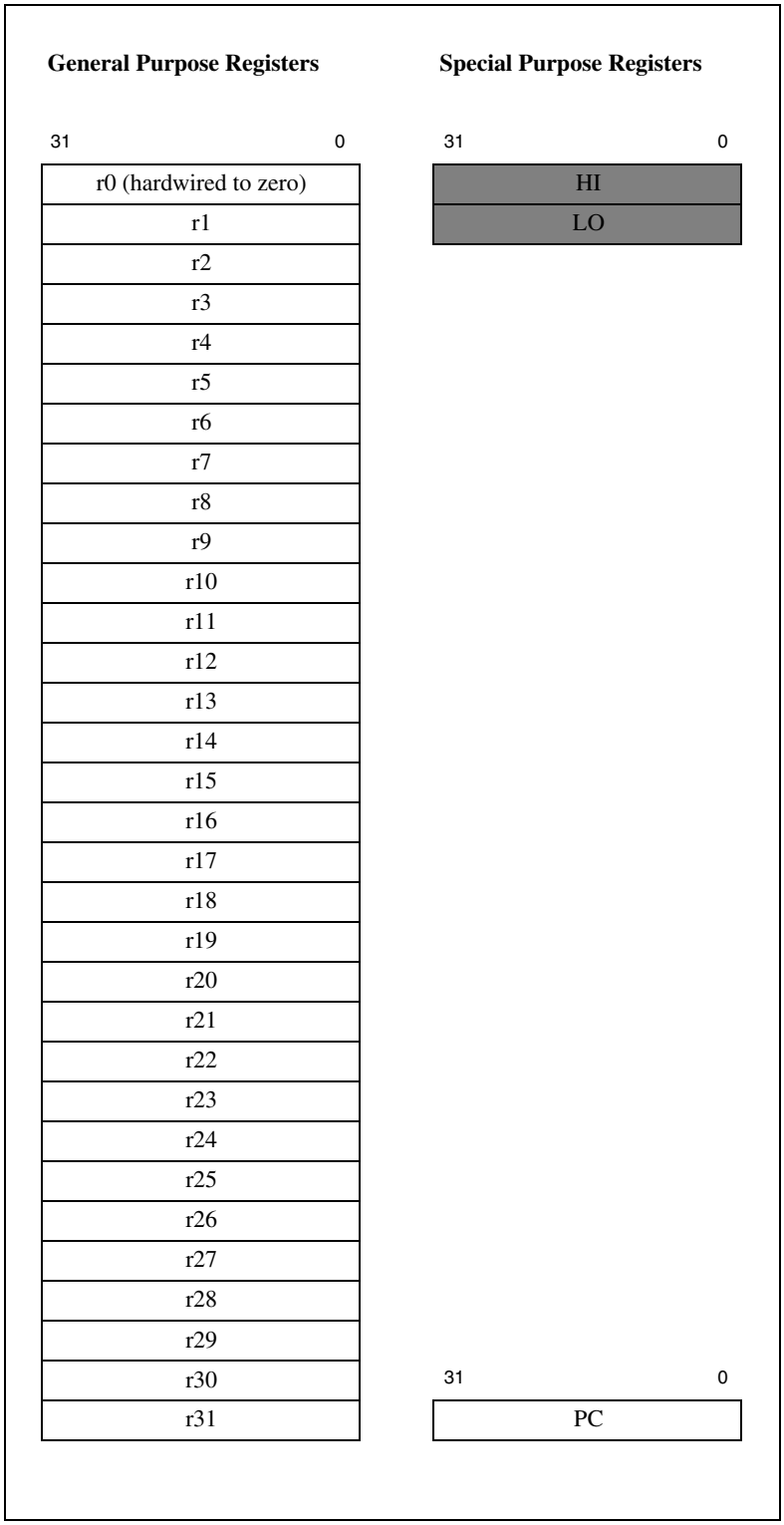
- *PC*—Program Counter register
- *HI*—Multiply and Divide register higher result (removed in Release 6)
- *LO*—Multiply and Divide register lower result (removed in Release 6)
    - During a multiply operation, the *HI* and *LO* registers store the product of integer multiply.
    - During a multiply-add 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.

As of Release 6, the *HI* and *LO* registers and related instructions are removed from the base instruction set architecture and only exist in the DSP Module.

Figure 4.1 shows the layout of the CPU registers for MIPS32. The HI and LO registers, removed in Release 6, are shown as shaded.

Figure 4.1 CPU Registers for MIPS32



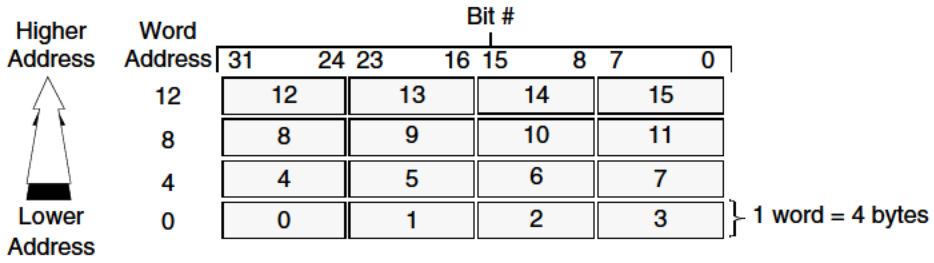
4.4 Byte Ordering and Endianness

Bytes within larger CPU data formats—halfword, word, and doubleword—can be configured in either big-endian or little-endian order. *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 4.2 and 4.3 show the ordering of bytes within words and the ordering of words within multiple-word structures for both big-endian and little-endian configurations.

4.4.1 Big-Endian Order

When configured in **big-endian order**, byte 0 is the most-significant (left-hand) byte. Figure 4.2 shows this configuration.

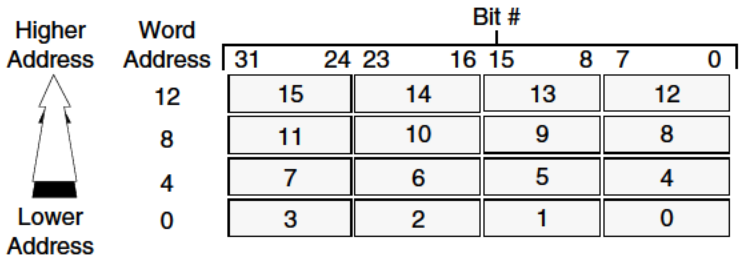
Figure 4.2 Big-Endian Byte Ordering



4.4.2 Little-Endian Order

When configured in **little-endian order**, byte 0 is always the least-significant (right-hand) byte. Figure 4.3 shows this configuration.

Figure 4.3 Little-Endian Byte Ordering



4.4.3 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. Big and Little Endian byte ordering for double words is shown in Figure 4.4 and Figure 4.5.

Figure 4.4 Big-Endian Data in Doubleword Format

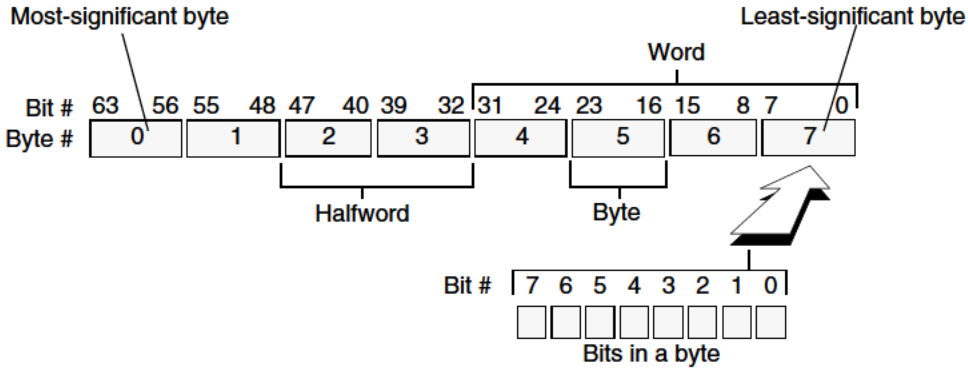
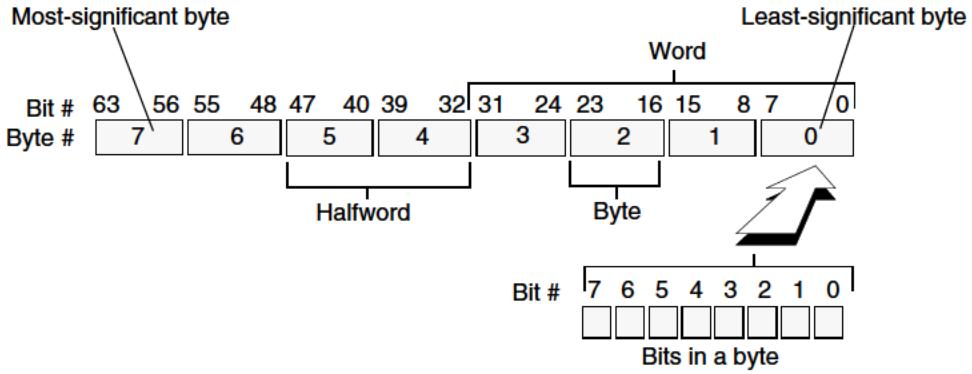


Figure 4.5 Little-Endian Data in Doubleword Format



4.5 Memory Alignment

Releases of the architecture prior to Release 5 required "natural" alignment of memory operands for memory operations, as described in section 4.5.1 "Addressing Alignment Constraints" below. Instructions such as LWL and LWR, described in section 4.5.2 "Unaligned Load and Store Instructions (Removed in Release 6)", were provided so that unaligned accesses could be performed by instruction sequences.

In Release 5 of the Architecture, the MSA (MIPS SIMD Architecture) supports 128-bit memory accesses and does NOT require these accesses to be naturally aligned.

Release 6 requires misaligned memory accesses to be supported by all Release 6-compliant implementations for all ordinary memory instructions and removes the unaligned load and store instructions. The Release 6 requirement to support misaligned memory accesses using trap and emulate is system-level, not necessarily hardware-level. Because misaligned memory accesses may be slow, it is recommended that software use naturally aligned memory accesses whenever possible.

The behavior, semantics, and architecture specifications of such misaligned accesses are described in Appendix B, "Misaligned Memory Accesses" on page 128.

4.5.1 Addressing Alignment Constraints

In MIPS architectures prior to Release 6, the following natural alignment constraints apply to the byte addresses for halfword, word, and doubleword accesses:

- 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...).

This natural alignment extends to larger data accesses, such as 128-bit MSA vectors.

In Release 6, misaligned accesses are permitted, though the natural alignment constraints listed above are recommended for best performance.

4.5.2 Unaligned Load and Store Instructions (Removed in Release 6)

In architectures prior to Release 6, the instructions listed in Table 4.1 are used to load and store words that are not aligned on word or doubleword boundaries. These instructions are removed in Release 6.

Table 4.1 Unaligned Load and Store Instructions

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

Figure 4.6 shows a big-endian access of a misaligned word that has byte address 3, and Figure 4.7 shows a little-endian access of a misaligned word that has byte address 1. Both figures show left-side misalignment.

Figure 4.6 Big-Endian Misaligned Word Addressing

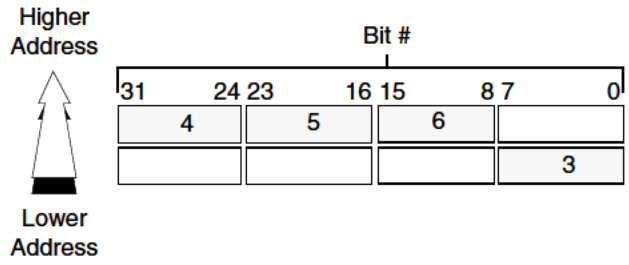
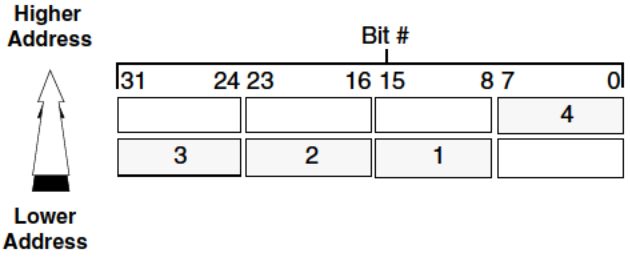


Figure 4.7 Little-Endian Misaligned Word Addressing



4.6 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

Implementations may include this optional memory access type:

- Uncached Accelerated

4.6.1 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.

4.6.2 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 that may never be executed is allowed.

4.6.3 Uncached Accelerated Memory Access

In revisions of this specification prior to 3.50, the behavior of the Uncached Accelerated Memory Access type was not architecturally defined, but rather was implementation-specific. Beginning with the 3.50 revision, the behavior of the Uncached Accelerated access type is defined, and the access type is optional.

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

In uncached access, each store instruction causes a separate, unique request to physical memory.

In MIPS CPUs, writes are allowed to be buffered within the CPU. Write buffers are usually of cache-line in size. Usually, if there is sufficient data within the write buffer, the data is sent in one burst transaction for higher efficiency.

In uncached accelerated access, the data from multiple store instructions can be sent together to the physical memory in one burst transaction. This is achieved by using write buffers to gather the data from multiple store instructions before sending out the memory request.

Data from store instructions using uncached accelerated access are kept in the buffer according to the following rules:

- Buffering can start on any byte address.
- Data is placed into the buffer obeying full byte addressing.
- Data is placed into the buffer for any request size - byte, halfword, word, doubleword, and the 3, 5-7 byte sizes allowed by unaligned or misaligned memory accesses.
- A byte can be overwritten with new data before the buffer data is flushed out of the core.
- Multiple buffers (each holding data from multiple store instructions) can be active at one time.

The uncached accelerated data within the write-buffer is sent to physical memory according to the following rules:

- As a consequence of the execution of a SYNC instruction. All uncached accelerated data within all write buffers is sent to physical memory in this situation.
- If a write-buffer is entirely full with uncached accelerated data. Typically, an entire cache line of UCA data is emptied to physical memory.
- If the target address of any load instruction matches the address of any uncached accelerated data within the write buffer.
- If the target address of any store instruction using any other type of access type matches the address of any uncached accelerated data within the write buffer.
- As a consequence of the execution of a non-coherent SYNCI instruction. All uncached accelerated data within all write buffers is sent to physical memory in this situation.
- If the target address of a PREF Nudge operation matches the address of any uncached accelerated data within the write buffer.
- All write buffers capable of holding uncached accelerated data are already active, and another store instruction using uncached accelerated access is executed whose target address does not match any of these write-buffers. In this case, at least one of the write buffers must be emptied to physical memory to make space for the new store data.

4.7 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.

4.8 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) generated by the MMU for the access.

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.

4.9 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 instruction-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.

4.10 Instruction Fetch

4.10.1 Instruction Fields

For MIPS instructions, the layout of the bit fields in instructions is little-endian, regardless of the endianness mode in which the processor is executing. Bit 0 of an instruction is always the right-most bit in the instruction, while bit 31 is always the left-most bit in the instruction. The major opcode is always the left-most 6 bits in the instruction.

4.10.2 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 4.8 shows an example where the width of external memory is 64 bits (two words), 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 right-most word of the dword, while the more-significant address is the left-most word within the dword.

Figure 4.8 Two Instructions Placed in a 64-bit Wide, Little-endian Memory

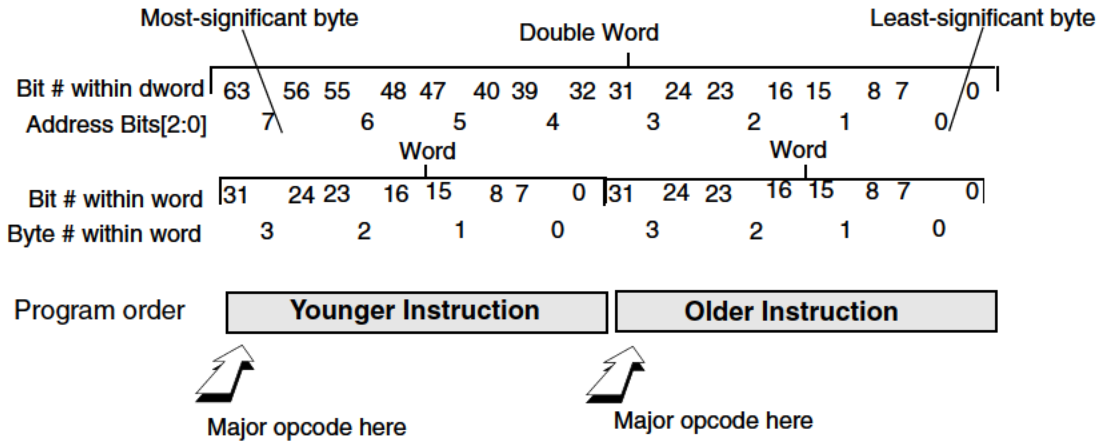
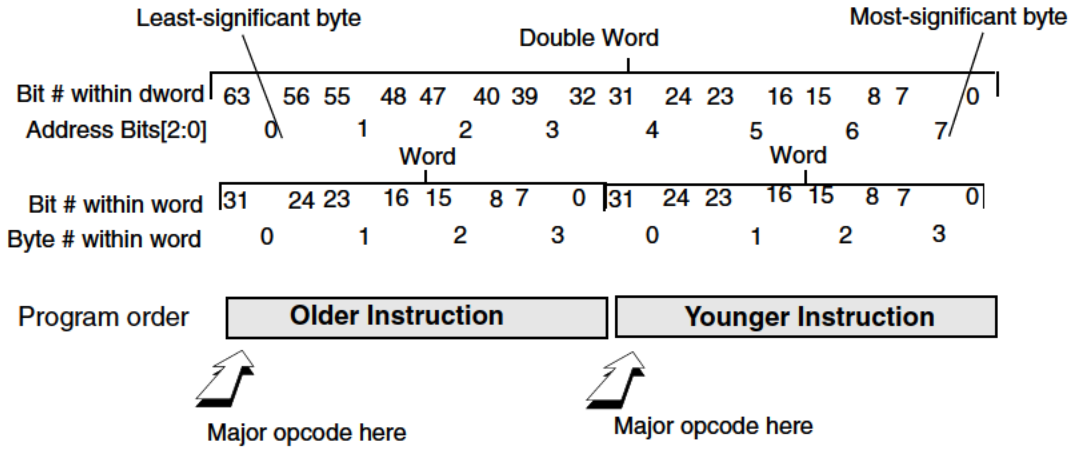


Figure 4.9 shows the equivalent big-endian example where the less-significant address refers to the left-most word within the dword, and the more-significant address refers to the right-most word within the dword. In both BE and LE examples, the bit locations within the instruction words has not changed. The location of the major opcode is always in the left-most bits within the word.

Figure 4.9 Two instructions Placed in a 64-bit Wide, Big-endian Memory



4.10.3 Instruction Fetch Using Uncached Access Without Side-effects

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

The fetch transfer size for uncached memory access may be larger than one instruction word. In this case, it is implementation-specific whether there are multiple instruction fetches to the same memory location. It is not required that the processor has a register to buffer the unused instructions of the transfer for subsequent execution.

Speculative instruction fetches are allowed. Types of speculative instruction fetches are listed in Table 4.2.

In Release 5, CP0 *MAAR* and *MAARI* allow software to specify which address regions are speculatable, in combination with the CCA (Cacheability and Coherency Attribute) of the access.

Table 4.2 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 the 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.

4.10.4 Instruction Fetch Using Uncached Access With Side-effects

Side-effects of accesses for a memory region might include FIFO behavior, stack behavior, or location-specific behavior (where one memory location defines 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 EJTAG Debug Memory space (dmseg) is defined by the architecture as having memory access side-effects. Refer to *MIPS® EJTAG Specification*, MIPS Document MD00047.

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 that the architecture does not define to have access side-effects. For that reason, any implementation that allows behaviors listed in 4.10.3 "Instruction Fetch Using Uncached Access Without Side-effects" should restrict software from executing code within any memory region with side-effects.

In Release 5, CP0 *MAAR* and *MAARI* allow software to specify which address regions are speculatable, in combination with the CCA (Cacheability and Coherency Attribute) of the access.

4.10.5 Instruction Fetch Using Cacheable Access

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 4.2 list some types of speculative instruction fetches.

In Release 5, CP0 *MAAR* and *MAARI* allow software to specify which address regions are speculatable, in combination with the CCA (Cacheability and Coherency Attribute) of the access.

4.10.6 Instruction Fetches and Exceptions

4.10.6.1 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 privilege) for that not-to-be-executed instruction may not generate any exception.

4.10.6.2 Instruction Fetch Exceptions on Branch Delay Slots and Forbidden 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.

This consideration does NOT necessarily apply to implementations of branch forbidden slots. An exception on a branch forbidden slot is not reported if the branch is taken. If the branch is not taken, implementations have the option of reporting the exception on the branch forbidden slot with the restart PC of the forbidden slot or of the branch. Software should allow for either implementation.

4.10.7 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 that is shared by both the instruction fetches 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 that 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 5

CPU Instruction Set

This chapter provides an overview of the CPU instruction set. The instructions are organized into the following functional groups:

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

CPU instructions are listed and described by type in Table 5.1 Table 5.1 through Table 5.28Table 5.28. The instruction tables specify the MIPS architecture ISA(s) in which the instruction is defined—for example, "MIPS32" indicates that the operation is present in all revisions of MIPS32, "MIPS64, MIPS32 Release 2" indicates that the operation is present in all versions of MIPS64 and is present in MIPS32 Release 2 and all later versions of MIPS32, unless otherwise noted; "Release 6" indicates that the operation is present in Release 6 and not in previous revisions; "Removed in Release 6" means that implementations of Release 6 are required to signal the Reserved Instruction exception when there is no higher priority exception, and when the instruction encoding has not been reused for a different instruction

5.1 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.

5.1.1 Types of Loads and Stores

There are several 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)[1]
- Selecting the addressing mode (for example, SDXC1, in the FPU)
- Atomic memory update (read-modify-write: for instance, LL/SC)
- PC-relative loads (e.g., LWPC) (Release 6 only)

1. Release 6 removes the unaligned memory access instructions (e.g., LWL/LWR), requiring support for misaligned memory accesses for all ordinary memory operations.

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 43 for more information on big-endian and little-endian data ordering.

5.1.2 Load and Store Access Types

Table 5.1 lists Table 5.1lists the data sizes that can be accessed by CPU load and store operations.

Table 5.1 Load and Store Operations

Data Size	CPU			Coprocessors 1 and 2	
	Load Signed	Load Unsigned	Store	Load	Store
Register + Offset Addressing Mode					
Byte	MIPS32	MIPS32	MIPS32		
Halfword	MIPS32	MIPS32	MIPS32		
Word	MIPS32	MIPS64	MIPS32	MIPS32	MIPS32
Doubleword (FPU)				MIPS32	MIPS32
Unaligned word (e.g., LWL/ LWR)	MIPS32 Removed in Release 6		MIPS32 Removed in Release 6		
Linked word (atomic modify)	MIPS32		MIPS32		
PC-relative Addressing Mode					
PC-relative word	Release 6				
FPU Load and Store Operations Using Register + Register Addressing Mode					
Word (LWXC1, SWXC1)				MIPS64 MIPS32 Release 2 Removed in Release 6	
Doubleword (LDXC1, SDXC1)					
Unaligned Doubleword Indexed (LUXC1, SUXC1)					

5.1.3 List of CPU Load and Store Instructions

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

- Byte
- Halfword
- Word

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

Table 5.2 lists CPU load and store instructions that required natural alignment prior to MIPS Release 6 (refer to Section 4.5.1 "Addressing Alignment Constraints") Note that in Release 6, natural alignment for these instructions is is no longer required.

Table 5.2 Naturally Aligned CPU Load/Store Instructions

Mnemonic	Instruction	Defined in MIPS ISA
LB	Load Byte	MIPS32
LBE	Load Byte EVA	MIPS32 Release 5
LBU	Load Byte Unsigned	MIPS32
LBUE	Load Byte Unsigned EVA	MIPS32 Release 5
LH	Load Halfword	MIPS32
LHE	Load Halfword EVA	MIPS32 Release 5
LHU	Load Halfword Unsigned	MIPS32
LHUE	Load Halfword Unsigned EVA	MIPS32 Release 5
LW	Load Word	MIPS32
LWE	Load Word EVA	MIPS32 Release 5
SB	Store Byte	MIPS32
SBE	Store Byte EVA	MIPS32 Release 5
SH	Store Halfword	MIPS32
SHE	Store Halfword EVA	MIPS32 Release 5
SW	Store Word	MIPS32
SWE	Store Word EVA	MIPS32 Release 5

In architectures prior to Release 6, unaligned words and doublewords can be loaded or stored using a pair of the special instructions listed in Table 5.3. The load instructions read the left-side or right-side bytes (left or right side of register) from an aligned word and merge them into the correct bytes of the destination register. These instructions are removed in Release 6.

Table 5.3 Unaligned CPU Load and Store Instructions

Mnemonic	Instruction	Defined in MIPS ISA
LWL	Load Word Left	MIPS32 Removed in Release 6
LWLE	Load Word Left EVA	MIPS32 Removed in Release 6
LWR	Load Word Right	MIPS32 Removed in Release 6
LWRE	Load Word Right EVA	MIPS32 Removed in Release 6
SWL	Store Word Left	MIPS32 Removed in Release 6

Table 5.3 Unaligned CPU Load and Store Instructions (Continued)

Mnemonic	Instruction	Defined in MIPS ISA
SWLE	Store Word Left EVA	MIPS32 Removed in Release 6
SWR	Store Word Right	MIPS32 Removed in Release 6
SWRE	Store Word Right EVA	MIPS32 Removed in Release 6

5.1.3.1 PC-relative Loads (Release 6)

The Release 6 ISA provides the following PC-relative loads with a span of +/- 1 Megabytes:

- LWPC: Loads a 32-bit word from a PC-relative address, formed by adding the word-aligned PC to a sign-extended 19-bit immediate shifted left by 2 bits, giving a 21-bit span.
- LWUPC: Loads a 32-bit unsigned word from a PC-relative address, formed by adding the word-aligned PC to a sign-extended 19-bit immediate shifted left by 3 bits, giving a 21-bit span.
- LDPC: Loads a 64-bit doubleword from a PC-relative address, formed by adding the PC, aligned to 8 bytes by masking off the low 3 bits, to a sign-extended 18-bit immediate, shifted left by 3 bits, giving a 21-bit span.

Note that PC-relative load instructions can only generate aligned memory addresses.

Table 5.4 PC-relative Loads

Mnemonic	Instruction	Defined in MIPS ISA
LWPC	Load Word, PC-relative	MIPS32 Release 6

5.1.4 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. LL/SC family instructions require alignment, even in Release 6, where ordinary instructions do not.

Table 5.5 Atomic Update CPU Load and Store Instructions

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

5.1.5 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 5.6 lists the coprocessor load and store instructions. In Release 6, Coprocessor 1 (FPU) instructions are the same as in previous releases, with a 16-bit offset, but the coprocessor 2 instructions LWC2/SWC2 and LDC2/SDC2 change encodings, and have only an 11-bit offset.

Table 5.6 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 5.7 lists the FPU indexed load and store instructions.[2]

Table 5.7 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 Removed in Release 6
SWXC1	Store Word Indexed from Floating Point	
LDXC1	Load Doubleword Indexed to Floating Point	
SDXC1	Store Doubleword Indexed from Floating Point	
LUXC1	Load Doubleword Indexed Unaligned to Floating Point	
SUXC1	Store Doubleword Indexed Unaligned from Floating Point	

5.2 Computational Instructions

This section describes the following types of instructions:

- ALU Immediate and Three-Operand Instructions
- ALU Two-Operand Instructions
- Shift Instructions
- Width Doubling Multiply and Divide Instructions (Removed in Release 6)

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

2. For convenience, FPU indexed loads and stores are listed here with the other coprocessor loads and stores.

The add and subtract operations labeled “unsigned” are actually modulo arithmetic without overflow detection. The “signed” add and subtract instructions detect overflow beyond the limits of a signed 2’s complement integer.

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.

5.2.1 ALU Immediate and Three-Operand Instructions

Table 5.8 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.

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.

Release 6 removes the add instructions with a large immediate field that trap on signed overflow (ADDI, DADDI), but retains the register-register forms (ADD, DADD).

Release 6 includes additional instructions with large constants, as described in Section 5.4 “Address Computation and Large Constant Instructions (Release 6)”.

Table 5.8 ALU Instructions With a 16-bit Immediate Operand

Mnemonic	Instruction	Defined in MIPS ISA
ADDI	Add Immediate Word	MIPS32, Removed in Release 6
ADDIU[1]	Add Immediate Unsigned Word	MIPS32
ANDI	And Immediate	MIPS32
LUI	Load Upper Immediate	MIPS32
ORI	Or Immediate	MIPS32
SLTI	Set on Less Than Immediate	MIPS32
SLTIU	Set on Less Than Immediate Unsigned	MIPS32
XORI	Exclusive Or Immediate	MIPS32

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 5.9 describes ALU instructions that use three operands.

Table 5.9 Three-Operand ALU Instructions

Mnemonic	Instruction	Defined in MIPS ISA
ADD	Add Word	MIPS32
ADDU[1]	Add Unsigned Word	MIPS32
AND	And	MIPS32
NOR	Nor	MIPS32
OR	Or	MIPS32
SLT	Set on Less Than	MIPS32
SLTU	Set on Less Than Unsigned	MIPS32

Table 5.9 Three-Operand ALU Instructions (Continued)

Mnemonic	Instruction	Defined in MIPS ISA
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.

5.2.2 ALU Two-Operand Instructions

Table 5.10 describes ALU instructions that use two operands.

Table 5.10 Two-Operand ALU Instructions

Mnemonic	Instruction	Defined in MIPS ISA
CLO	Count Leading Ones in Word	MIPS32
CLZ	Count Leading Zeros in Word	MIPS32

5.2.3 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 example, SLL, SRL)
- Those that take a shift amount from the low-order bits of a general register (for example, SRAV, SRLV)

The Release 6 ISA defines the BITSWAP instruction, which reverses the bits in every byte of its operand. Release 6 BITSWAP corresponds to MIPS DSP Module BITREV.

The Release 6 ISA defines the ALIGN instruction that concatenates its two operands and the same-width contiguous subset from the concatenation at byte granularity. ALIGN is useful for extracted data at a misaligned memory address from two aligned memory load results. Release 6 ALIGN corresponds to MIPS DSP Module BALIGN.

Shift instructions are listed in Table 5.11.

Table 5.11 Shift Instructions

Mnemonic	Instruction	Defined in MIPS ISA
ALIGN	Extract byte-aligned word from concatenation of two words	MIPS32 Release 6
BITSWAP	Swap bits in every byte of word operand	MIPS 32 Release 6
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

Table 5.11 Shift Instructions (Continued)

Mnemonic	Instruction	Defined in MIPS ISA
SRLV	Shift Word Right Logical Variable	MIPS32

5.2.4 Width Doubling Multiply and Divide Instructions (Removed in Release 6)

- The width doubling 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. In Release 6, these instructions are moved to the DSP Module. **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 5.12 lists the multiply, divide, and *HI*/*LO* move instructions.

Table 5.12 Multiply/Divide Instructions

Mnemonic	Instruction	Defined in MIPS ISA
DIV	Divide Word	MIPS32 Removed in Release 6
DIVU	Divide Unsigned Word	MIPS32 Removed in Release 6
MADD	Multiply and Add Word	MIPS32 Removed in Release 6
MADDU	Multiply and Add Word Unsigned	MIPS32 Removed in Release 6
MFHI	Move From HI	MIPS32 Removed in Release 6
MFLO	Move From LO	MIPS32 Removed in Release 6
MSUB	Multiply and Subtract Word	MIPS32 Removed in Release 6
MSUBU	Multiply and Subtract Word Unsigned	MIPS32 Removed in Release 6
MTHI	Move To HI	MIPS32 Removed in Release 6
MTLO	Move To LO	MIPS32 Removed in Release 6
MUL	Multiply Word to Register	MIPS32 Removed in Release 6
MULT	Multiply Word	MIPS32 Removed in Release 6

Table 5.12 Multiply/Divide Instructions (Continued)

Mnemonic	Instruction	Defined in MIPS ISA
MULTU	Multiply Unsigned Word	MIPS32 Removed in Release 6

5.2.5 Same-Width Multiply and Divide Instructions (Release 6)

The Release 6 multiply and divide instructions produce results that are the same width as their operands, using ordinary GPRs as both input and output. They are available in 32-bit and 64-bit, signed and unsigned arithmetic.

- **Multiply-low** instructions (MUL, MULU, DMUL, DMULU) produce the low 32-bits or 64-bit of the product.
- **Multiply-high** instructions (MUH, MUHU, DMUH, DMUHU) produce the high 32-bits or 64-bit of the product.

Note that the low half of a product is the same for signed and unsigned 2's-complement multiplication, but the upper half differs, for example, MUL and MULU produce the same result, but MUH and MUHU produce different results.

- **Divide** instruction produce a quotient that is loaded into a single GPR destination (DIV, DIVU, DDIV, DDIVU)
- **Modulus** instructions produce a remainder that is loaded into a single GPR destination (MOD, MODU, DMOD, DMODU)

If the full double-width product is desired for a multiplication, or both quotient and remainder are desired for a division, the appropriate same-width instructions should be used in close proximity to permit hardware optimizations. For example, the multiply-low instruction MULU may retain its result, to be provided to the following multiply-high MUHU instruction.

Table 5.13 lists the same-width multiply and divide instructions.

Table 5.13 Same-width Multiply/Divide Instructions (Release 6)

Mnemonic	Instruction	Defined in MIPS ISA
MUL	Multiply word, Low part, signed	MIPS32 Release 6
MUH	Multiply word, High part, signed	MIPS32 Release 6
MULU	Multiply word, How part, Unsigned	MIPS32 Release 6
MUHU	Multiply word, High part, Unsigned	MIPS32 Release 6
DMUL	Multiply doubleword, Low part, signed	MIPS64 Release 6
DMUH	Multiply doubleword, High part, signed	MIPS64 Release 6
DMULU	Multiply doubleword, How part, Unsigned	MIPS64 Release 6
DMUHU	Multiply double word, High part, Unsigned	MIPS64 Release 6
DIV	Divide words, signed	MIPS32 Release 6
MOD	Modulus remainder word division, signed	MIPS32 Release 6
DIVU	Divide words, Unsigned	MIPS32 Release 6
MODU	Modulus remainder word division, Unsigned	MIPS32 Release 6
DDIV	Divide doublewords, signed	MIPS64 Release 6
DMOD	Modulus remainder doubleword division, signed	MIPS64 Release 6
DDIVU	Divide doublewords, Unsigned	MIPS64 Release 6

Table 5.13 Same-width Multiply/Divide Instructions (Release 6)

Mnemonic	Instruction	Defined in MIPS ISA
DMODU	Modulus remainder doubleword division, Unsigned	MIPS64 Release 6

5.3 Jump and Branch Instructions

This section describes the following:

- Types of Jump and Branch Instructions
- Branch Delay Slots and Branch Likely versus Compact Branches and Forbidden Slots
- Jump and Branch Instructions

5.3.1 Types of Jump and Branch Instructions

The architecture defines the following jump and branch instructions:

- PC-relative conditional branch
- PC-region unconditional jump
- PC-relative unconditional branch
  - Release 6 introduces unconditional branches with 26-bit offsets.
- Absolute (register) unconditional jumps
  - Register-indirect jumps
  - Register indexed jumps (Release 6)
- Procedures
  - A set of procedure calls that record a return link address in a general register.
  - Procedure return is performed by register indirect jumps that use the contents of the link register, r31, as the branch target. This provides a hint to the hardware to adjust the call/return predictor appropriately.

5.3.2 Branch Delay Slots and Branch Likely versus Compact Branches and Forbidden Slots

The original MIPS architecture supports, indeed requires, delayed branches: the instruction after the branch may be executed before the branch is taken (since typically the instruction after the branch has already been fetched). Code optimization can place instructions into this branch delay slot for improved performance. Typical delayed branches execute the delay-slot instruction, whether or not the branch was taken.

Branch-likely instructions execute the delay-slot instruction if and only if the branch is taken. If the branch is not taken, the delay-slot instruction is not executed. While Branch Likely was deprecated prior to Release 6, Release 6 removes Branch Likelies.

Release 6 introduces conditional compact branches and compact jumps that do not have a delay slot; they have instead a forbidden slot. Release 6 unconditional compact branches have neither a delay slot nor a forbidden slot.

5.3.2.1 Control Transfer Instructions in Delay Slots and Forbidden Slots

In MIPS architectures prior to Release 6. if a control transfer instruction (CTI) is placed in a branch delay slot, the operation of both instructions is **UNPREDICTABLE**. In Release 6, if a CTI is placed in a branch delay slot or a compact branch forbidden slot, implementations are required to signal a Reserved Instruction exception.

The following instructions are forbidden in branch delay slots and forbidden slots: any CTI, including branches and jumps, ERET, ERETNC, DERET, WAIT, and PAUSE. Their occurrence is required to signal a Reserved Instruction exception.

5.3.2.2 Exceptions and Delay and Forbidden Slots

If an exception or interrupt prevents the completion of an instruction in a delay slot or forbidden slot, the hardware reports an Exception PC (CP0 *EPC*) of the branch, with a status bit set ($Status_{BD}$) to indicate that the exception was for the instruction in the delay slot of the branch.

By convention, if an exception or interrupt prevents the completion of an instruction in a branch delay or forbidden slot, the branch instruction is re-executed and branch instructions must be restartable to permit this. In particular, procedure calls must be restartable. To insure that a procedure call is restartable, procedure calls that have a delay slot or forbidden slot (branch/jump-and-link instructions) should not use the register in which the return link is stored (usually GPR 31) as a source register. This applies to all branch/jump-and-link instructions that have both a slot (delay or forbidden) and source registers, both for conditions (e.g., BGEZAL or BGEZALC) or for jump targets (JALR). This does not apply to branch/jump-and-link instructions that do not have source registers (e.g., BAL, JAL, BALC). Nor does it apply to the Release 6 unconditional compact jump-and-link (JIALC) instruction, which has a source register but has neither a delay slot nor a forbidden slot.

5.3.2.3 Delay Slots and Forbidden Slots Performance Considerations

Delay slots and forbidden slots are dynamic, not static; that is, a branch instruction immediately following another branch instruction is not necessarily in a delay or forbidden slot. Nevertheless, although adjacent control transfer instructions are allowed in some situations, the hardware implementation often imposes a performance penalty on them.

To optimize code, software should avoid generating code with adjacent CTIs, at least in the specific cases known to cause performance problems, namely, two adjacent CTIs within the same 64-bit aligned block of instruction memory.[3] Also, compilers should avoid placing data (that may look like a CTI if speculatively decoded as an instruction) in the same 64-bit aligned block of instruction memory as an actual CTI. This constraint applies to processors with hardware branch prediction that assumes the presence of only 1 CTI in a 64-bit aligned block of memory to minimize storage requirements.

Keep in mind that implementations may have performance constraints that expand on the simple rules mentioned here. For example, an implementation may have a performance problem if there are two frequently executed taken branches within the same 16-byte aligned block of instruction memory. However, these problems are beyond the scope of this document.

3. Note: This rule is not the same as “no adjacent branches”, or “no branches in delay slots”.

5.3.2.4 Examples of Delay Slots and Forbidden Slots

The examples in this section describe legacy and new Release 6 compact branch and jump handling. The reader may interchange branch and jump in the examples, where applicable.

5.3.2.4.1 Statically Adjacent Branches (Generic Handling)

```
Address A:    Branch1
Address A+4:  Branch2
...
Address AA:   Branch3 to A+4
```

In this example, Branch2 is statically adjacent to and below Branch1. However, Branch2 is not necessarily in Branch1's delay or forbidden slot. If Branch2 is reached by a non-adjacent branch from Branch3, this has defined behavior, and a Reserved Instruction exception must not be signalled. Branch2 is only in Branch1's delay or forbidden slot if Branch1 is actually executed, and the semantics of Branch1 are such that the delay/forbidden slot (Branch2), must be executed before the instruction at the target of Branch1. The Release 6 Reserved Instruction exception for CTIs in forbidden slots must only be signalled if actually executed.

5.3.2.4.2 Statically Adjacent Delay Slot Branches (Legacy Handling)

```
0000F000:     BGEZ r1,target1
0000F004:     BLTZ r2,target2
```

In this example, the branch at address F004 is not necessarily in a delay slot. But if the branch at address F000 is executed, the branch at address F004 is in the delay slot whether or not the earlier branch is taken or not, and a Release 6 implementation must signal a Reserved Instruction exception. This is because the semantics of BGEZ require that the instruction in the delay slot be executed regardless of whether BGEZ is taken or not.

The example code may be a performance problem even if not a correctness problem, since there are two adjacent CTIs in the same aligned block of memory.

5.3.2.4.3 No Forbidden Slot - Statically Adjacent Compact Branches (Release 6 Compact Branches)

```
0000F000:     BC target1
0000F004:     BLTZC r2,target2
```

In this example, the branch at address F0004 is never in a forbidden slot, because the statically preceding branch F000 is an unconditional compact branch that has no forbidden slot.

The example code may be a performance problem, even if not a correctness problem, since there are two adjacent CTIs in the same aligned block of memory.

5.3.2.4.4 Forbidden Slot Only if Not Taken - Statically Adjacent Compact Branches (Release 6 Compact Branches)

```
0000F000:     BGEZC r1,target1
0000F004:     BLTZC r2,target2
```

In this example, the branch at address F004 is not necessarily in a forbidden slot. Even if the branch at address F000 is executed, but is always taken, the branch at address F004 is never in a forbidden slot, and a Reserved Instruction exception must not be signalled. A Reserved Instruction exception must be signalled only if the branch at F000 is executed and not taken.

The example code may be a performance problem, even if not a correctness problem, because there are two adjacent CTIs in the same aligned block of memory.

5.3.2.5 Deprecation of Branch Likely Instructions

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).

Branch Likely instructions have been deprecated since before Release 1 of the MIPS32 architecture. Programmers have been strongly encouraged to avoid the use of the Branch Likely instructions, and were warned that these instructions would be removed from future revisions of the MIPS Architecture. The Branch Likely instructions are removed as of microMIPS and Release 6.

5.3.3 Jump and Branch Instructions

5.3.3.1 Release 6 Compact Branch and Jump Instructions

Most branches and jumps have 16-bit offsets. Branch offsets are scaled by 4, for a span of +/- 128KB. Jump offsets are not scaled, for a span of +/-32KB. Indexed Jumps add the unscaled sign-extended offset to their register argument. Not scaling allows tag bits to be eliminated from pointers, as are used in some dynamically typed languages.

The ISA provides a small set of control transfer instructions with larger spans. Unconditional branch (BC) and call (BALC) have 26-bit scaled spans, +/- 128MB. BEQZC and BNEZC operate on C-style truth values, and have 21 bit spans, +/- 4MB.

In addition to BEQZC and BNEZC, Release 6 defines a complete set of compare-and-branch comparisons of 16-bit span, +/- 128KB: compares between registers, both signed and unsigned, as well as comparisons against zero. The latter are needed because Release 6's encodings for the register-to-register compare and branch instructions are very tight, and do not allow GPR[0] to be specified as a register operand in many cases.

Release 6 also defines BOVC and BNVC, which branch according to whether the sum of their 32-bit signed inputs overflows (BOVC) or does not overflow (BNVC).

Table 5.14 Release 6 Compact Branch and Jump Instructions (Release 6)

Offset	Span	Mnemonic	Instruction	Defined in MIPS ISA
Unconditional Branch and Call				
26	**+/- 128MB**	BC	Compact Branch	MIPS32 Release 6
		BALC	Compact Branch And Link	MIPS32 Release 6
Indexed Jumps (register + unscaled offset)				
16	+/-32K	JIC	Compact Jump Indexed	MIPS32 Release 6
		JIALC	Compact Jump Indexed And Link	MIPS32 Release 6
Compare to Zero				
21	**+/- 4MB**	BEQZC	Compact Branch if Equal to Zero	MIPS32 Release 6
		BNEZC	Compact Branch if Not Equal to Zero	MIPS32 Release 6
16	+/- 128KB	BLEZC	Compact Branch if Less Than or Equal to Zero	MIPS32 Release 6
		BGEZC	Compact Branch if Greater Than or Equal to Zero	MIPS32 Release 6
		BGTZC	Compact Branch if Greater Than Zero	MIPS32 Release 6
		BLTZC	Compact Branch if Less Than Zero	MIPS32 Release 6
Conditional calls, compare against zero				
16	+/- 128KB	BEQZALC	Compact Branch if Equal to Zero, And Link	MIPS32 Release 6
		BNEZALC	Compact Branch if Not Equal to Zero, And Link	MIPS32 Release 6
		BLEZALC	Compact Branch if Less Than or Equal to Zero, And Link	MIPS32 Release 6
		BGEZALC	Compact Branch if Greater Than or Equal to Zero, And Link	MIPS32 Release 6
		BGTZALC	Compact Branch if Greater Than Zero, And Link	MIPS32 Release 6
		BLTZALC	Compact Branch if Less Than Zero, And Link	MIPS32 Release 6
Compare equality reg-reg				
16	+/- 128KB	BEQC	Compact Branch if Equal	MIPS32 Release 6
		BNEC	Compact Branch if Not Equal	MIPS32 Release 6
Compare signed reg-reg				
16	+/- 128KB	BGEC	Compact Branch if Greater than or Equal	MIPS32 Release 6
		BLTC	Compact Branch if Less Than	MIPS32 Release 6
Compare Unsigned reg-reg				
16	+/- 128KB	BGEUC	Compact Branch if Greater than or Equal, Unsigned	MIPS32 Release 6
		BLTUC	Compact Branch if Less Than, Unsigned	MIPS32 Release 6
Aliases Obtained by Reversing Operands				
16	+/- 128KB	*BLEC*	*Compact Branch if Less Than or Equal*	MIPS32 Release 6
		BGTC	*Compact Branch if Greater Than*	MIPS32 Release 6
		BLEUC	*Compact Branch if Less than or Equal, Unsigned*	MIPS32 Release 6
		BGTUC	*Compact Branch if Greater Than, Unsigned*	MIPS32 Release 6
Branch if Overflow				
16	+/- 128KB	BOVC	Compact Branch if Overflow (word)	MIPS32 Release 6
		BNVC	Compact Branch if No overflow, word	MIPS32 Release 6

5.3.3.2 Delayed Branch instructions

Table 5.15 lists unconditional jump instructions that jump to a procedure call within the current 256MB-aligned region.

Table 5.16 lists instructions that jump to an absolute address held in a register.

Table 5.17 lists branch instructions that compare two registers before conditionally executing a PC-relative branch.

Table 5.18 lists branch instructions that test a register—compare with zero—before conditionally executing a PC-relative branch.

Table 5.19 lists the deprecated Branch Likely Instructions.

Table 5.15 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 Release3 Removed in Release 6

Table 5.16 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 Removed in Release 6[1]
JR.HB	Jump Register with Hazard Barrier	MIPS32 Removed in Release 6[1]

1. Release 6 removes JR.{HB} as a separate instruction, making it a special case of JALR.{HB} with source register GPR[0] operand.

Table 5.17 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 5.18 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 Removed in Release 6[1]

Table 5.18 PC-Relative Conditional Branch Instructions Comparing With Zero (Continued)

Mnemonic	Instruction	Defined in MIPS ISA
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 Removed in Release 6[1]

1. Release 6 removes BGEZAL and BLTZAL and they are required to signal a Reserved Instruction exception.

Table 5.19 Deprecated Branch Likely Instructions

Mnemonic	Instruction	Defined in MIPS ISA
BEQL	Branch on Equal Likely	MIPS32 Removed in Release 6
BGEZALL	Branch on Greater Than or Equal to Zero and Link Likely	MIPS32 Removed in Release 6
BGEZL	Branch on Greater Than or Equal to Zero Likely	MIPS32 Removed in Release 6
BGTZL	Branch on Greater Than Zero Likely	MIPS32 Removed in Release 6
BLEZL	Branch on Less Than or Equal to Zero Likely	MIPS32 Removed in Release 6
BLTZALL	Branch on Less Than Zero and Link Likely	MIPS32 Removed in Release 6
BLTZL	Branch on Less Than Zero Likely	MIPS32 Removed in Release 6
BNEL	Branch on Not Equal Likely	MIPS32 Removed in Release 6

5.4 Address Computation and Large Constant Instructions (Release 6)

The Release 6 ISA provides the instructions shown in Table 5.20 that are especially suited to address computations and the creation of large constants. Large constants can be formed efficiently using the upper bits with the 16-bit immediates available in most memory access and arithmetic instructions.

Table 5.20 Address Computation and Large Constant Instructions

Mnemonic	Instruction	Defined in MIPS ISA
LSA	Left Shift Add (Word)	MIPS32 Release 6
DLSA	Left Shift Add (Doubleword)	MIPS64 Release 6
AUI	Add Upper Immediate (Word)	MIPS32 Release 6
DAUI	Add Upper Immediate (Doubleword)	MIPS64 Release 6
DAHI	Add High Immediate (Doubleword)	MIPS64 Release 6
DATI	Add Top Immediate (Doubleword)	MIPS64 Release 6

Table 5.20 Address Computation and Large Constant Instructions (Continued)

Mnemonic	Instruction	Defined in MIPS ISA
ADDIUPC	Add Immediate Unsigned to PC	MIPS32 Release 6
AUIPC	Add Upper Immediate to PC	MIPS64 Release 6
ALUIPC	Add Upper Immediate to Aligned PC	MIPS64 Release 6

- Left Shift Add: LSA and DLSA add two registers, one of which is optionally shifted by a scaling factor from 1 to 4, corresponding to a scaling multiplication, e.g., by element size in an array, by 1, 2, 4, 8, or 16.

- Add Upper Immediate family: Adds an immediate value to a register. The immediate value is sign-extended and shifted by 16 bits (AUI/DAUI), 32 bits (DAHI), and 48 bits (DATI).

- PC-relative address computation: Adds an immediate value to the PC. ADDIUPC adds an immediate to the lower bits of the PC, AUIPC adds the immediate to the upper bits 16-31 of the PC. and ALUIPC adds the immediate to the upper bits of the PC, zeroing the low 16 bits of the PC.

  Note: These instructions sign-extend the 16-bit immediate. The “unsigned” in the name “Add Immediate Unsigned to PC” refers to not performing signed 32-bit overflow detection. Compare to ADD/ADDU.

See section 5.1.3.1 “PC-relative Loads (Release 6)” on page 55, for the related PC-relative load instructions.

5.5 Miscellaneous Instructions

Miscellaneous instructions include:

- Instruction Serialization (SYNC and SYNCI)
- Exception Instructions
- Conditional Move Instructions
- Prefetch Instructions
- NOP Instructions

5.5.1 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 5.21 lists the synchronization instructions.

Table 5.21 Serialization Instruction

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

5.5.2 Exception Instructions

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

- System call and breakpoint instructions, which cause unconditional exceptions (Table 5.22).
- Trap instructions, which cause conditional exceptions based on the result of a comparison (Table 5.23).
- Trap instructions which cause conditional exceptions based on the result of a comparison of a register value with an *immediate* value (Table 5.24). These instructions are removed in Release 6.

Table 5.22 System Call and Breakpoint Instructions

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

Table 5.23 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 5.24 Trap-on-Condition Instructions Comparing an Immediate Value

Mnemonic	Instruction	Defined in MIPS ISA
TEQI	Trap if Equal Immediate	MIPS32 Removed in Release 6
TGEI	Trap if Greater Than or Equal Immediate	MIPS32 Removed in Release 6
TGEIU	Trap if Greater Than or Equal Immediate Unsigned	MIPS32 Removed in Release 6

Table 5.24 Trap-on-Condition Instructions Comparing an Immediate Value (Continued)

Mnemonic	Instruction	Defined in MIPS ISA
TLTI	Trap if Less Than Immediate	MIPS32 Removed in Release 6
TLTIU	Trap if Less Than Immediate Unsigned	MIPS32 Removed in Release 6
TNEI	Trap if Not Equal Immediate	MIPS32 Removed in Release 6

5.5.3 Conditional Move Instructions

MIPS32 includes instructions to conditionally move one CPU general register to another, based on testing bit 0 of the value in a third general register. They are listed in Table 5.25.

Release 6 removes these instructions, replacing them by conditional select instructions that test C- compatible zero/nonzero value of a GPR and select a GPR or 0. These instructions are compatible with the truth values in the C language, and they have only two register inputs (the third input, 0, is implicit). They are listed in Table 5.26.

For the Release 6 floating-point conditional selects refer to Chapter 7, “FPU Instruction Set” on page 104.)

Table 5.25 CPU Conditional Move Instructions (Removed in Release 6)

Mnemonic	Instruction	Defined in MIPS ISA
MOVF	Move Conditional on Floating Point False	MIPS32 Removed in Release 6
MOVN	Move Conditional on Not Zero	MIPS32 Removed in Release 6
MOVT	Move Conditional on Floating Point True	MIPS32 Removed in Release 6
MOVZ	Move Conditional on Zero	MIPS32 Removed in Release 6

.

Table 5.26 CPU Conditional Select Instructions (Release 6)

Mnemonic	Instruction	Defined in MIPS ISA
SELEQZ	Select GPR rs if GPR rt is Equal to Zero, else select 0	MIPS32 Release 6
SELNEZ	Select GPR rs if GPR rt is Not Equal to Zero, else select 0	MIPS32 Release 6

5.5.4 Prefetch Instructions

The PREF and PREFX instructions, shown in Table 5.27, are used to indicate that memory is likely to be used in a particular way in the near future and should be prefetched into the cache. A *hint* field may indicate prefetch policies, such as which cache they are fetched into and whether reading or writing is intended. The PREF instruction uses

register+offset addressing, and the PREFX uses register+register addressing. The PREFX instruction is removed in Release 6.

Table 5.27 Prefetch Instructions

Mnemonic	Instruction	Addressing Mode	Defined in MIPS ISA
PREF	Prefetch	Register+Offset	MIPS32
PREFX	Prefetch Indexed	Register+Register	MIPS64 MIPS32 Release 2 Removed in Release 6

5.5.5 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, the SSNOP instruction takes up one issue cycle on any processor, including super-scalar implementations of the architecture. Release 6 removes the timing- and microarchitecture-dependent semantics of SSNOP, treating SSNOP like an ordinary NOP.

Table 5.28 NOP Instructions

Mnemonic	Instruction	Defined in MIPS ISA
NOP	No Operation	MIPS32
SSNOP	Superscalar Inhibit NOP Release 6 requires SSNOP to be implemented as a NOP with no special behavior.	MIPS32

5.6 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

5.6.1 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 5.29.

Table 5.29 Coprocessor Definition and Use in the MIPS Architecture

Coprocessor	MIPS32	MIPS64
CP0	System Control	System Control

Table 5.29 Coprocessor Definition and Use in the MIPS Architecture (Continued)

Coprocessor	MIPS32	MIPS64
CP1	FPU	FPU
CP2	Implementation-specific	
CP3	FPU (COP1X)[1]	FPU (COP1X)

1. In Release 1 of the MIPS32 Architecture, Coprocessor 3 was an implementation-specific coprocessor. In the MIPS64 Architecture, and in Release 2 of the MIPS32 Architecture (and subsequent releases), it is used exclusively for the floating point unit and is not available for implementation-specific use. Release 1 MIPS32 implementations are therefore encouraged not to use Coprocessor 3 as an implementation-specific coprocessor.

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.

5.6.2 System Control Coprocessor 0 (CP0)

The system controller for all MIPS processors is implemented as coprocessor 0 (CP0[4]), the **System Control Coprocessor**. It provides the processor control, memory management, and exception handling functions.

5.6.3 Floating Point Coprocessor 1 (CP1)

If a system includes a **Floating Point Unit**, it is implemented as coprocessor 1 (CP1). 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.

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

5.6.3.1 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 55.

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

5.7 CPU Instruction Formats

A CPU instruction is a single 32-bit word, aligned on a 32-bit boundary. The fields used in CPU instructions are shown in Table 5.30. The CPU instruction formats are described in Section 5.7.2 "CPU Instruction Field Formats".

The instruction formats presented here are an overview of what instruction fields may need to be decoded and/or extracted both in hardware (decoders) and software (disassemblers, emulators) for most instructions. See the individual instruction descriptions in Volume II of this document set for details and special cases.

Table 5.30 CPU Instruction Format Fields

Field	Description
opcode	6-bit primary operation code. The 6 most-significant bits of the instruction encoding.
function	6-bit function field used to specify functions within the primary opcode SPECIAL. The 6 least-significant bits of the instruction encoding.
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
sa	5-bit shift amount
immediate	A constant stored inside the instruction (as opposed to a constant separately in memory, that must be accessed using a load instruction). Unless further qualified, *immediate* typically refers to a 16-bit immediate occupying the least significant 16 bits of a 32-bit MIPS instruction. This 16-bit signed immediate is used for logical operands, arithmetic signed operands, load/store address byte offsets, and PC-relative branch signed instruction displacements. Some instructions have other immediate widths, e.g., 9-, 10-, 21-, and 26-bit offsets and displacements, and the 26-bit instr_index.
offset	An immediate constant in the instruction, used in forming a memory address or a PC-relative branch target. 16-bit offsets using the 16-bit immediate field are most common, although certain instructions have 9-, 18-, 19-, 21-, and 26-bit offsets.
instr_index	26-bit index shifted left two bits to supply the low-order 28 bits of the jump target address. Prior to Release 6 of the Architecture, this was the only example of a 26-bit immediate constant.

5.7.1 Advanced Instruction Encodings (Release 6)

MIPS Release 6 uses more advanced instruction encodings than previous releases. Release 6 uses the greater variety of field widths—shorter 9-bit offset for less frequently used instructions, and larger, 18, 19, 21, and 26 bits—to provide larger spans for certain important instructions. In addition, Release 6 uses instruction encoding techniques such as placing constraints on register operands, in cases such as BEQC r1,r2, target, where reversing the operands would be equivalent, and hence a waste of instruction encoding space. These advanced instruction encodings mean that the traditional tables of instruction formats are insufficient and unwieldy. The figures below have been updated with the most important Release 6 instruction formats. Full details can be obtained from the instruction descriptions in Volume II of this document set.

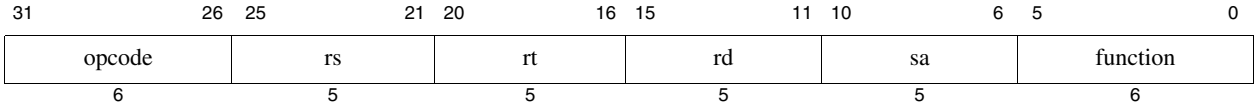
5.7.2 CPU Instruction Field Formats

The CPU instruction formats are shown in the figures below. The fields used in the instructions are described in Table 5.30.

Figures 5.1, 5.3, and 5.8 are the only formats applicable to releases of the architecture prior to Release 6. Figure 5.2 expands on additional formats used by Release 6, with Imm16 type shown as reference.

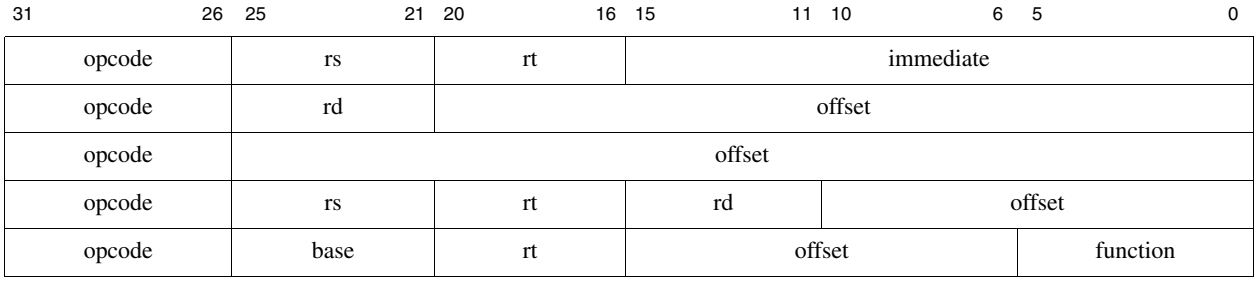
MIPS base ISA instructions have three, 5-bit fields suitable for specifying register numbers (rd, rs, rt) or a 5-bit shift amount (sa). Their bit positions are shown in Figure 5.1.

Figure 5.1 Register (R-Type) CPU Instruction Format



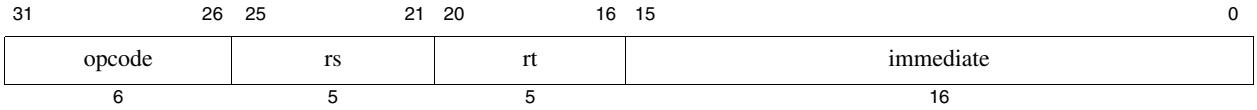
Several different Immediate (I-Type) instruction formats are shown in Figure 5.2. The 16-bit immediate constant inside the first instruction format can be used for both computation and memory/branch offset; the immediates in the other formats are mainly used as memory offset or branch displacement.

Figure 5.2 Immediate (I-Type) CPU Instruction Formats (Release 6)



The most common MIPS Immediate (I-Type) instruction format is the Imm16 format shown in Figure 5.3. The 16-bit signed immediate is used for logical operands, arithmetic signed operands, load/store address byte offsets, and PC-relative branch signed instruction displacements.

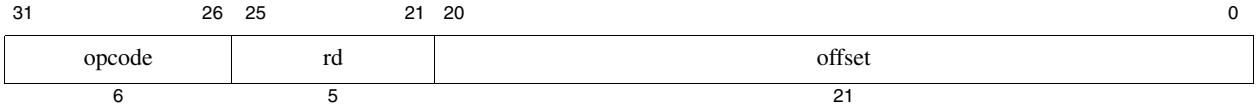
Figure 5.3 Immediate (I-Type) Imm16 CPU Instruction Format



MIPS Release 6 introduces the Immediate (I-Type) Off21 CPU instruction format, shown in Figure 5.3, used by instructions that compare a register against zero and branch (e.g., BLTZC), with larger than the usual 16-bit span.

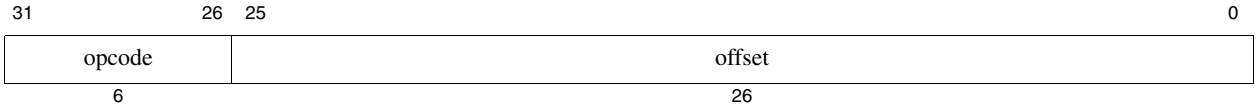
Certain PC-relative instructions use offsets 18- and 19-bits wide, using low bits of the 21-bit immediate as extra opcode bits.

Figure 5.4 Immediate (I-Type) Off21 CPU Instruction Format (Release 6)



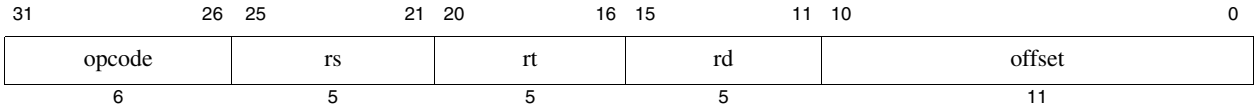
MIPS Release 6 introduces the Immediate (I-Type) Off26 CPU instruction format, shown in Figure 5.3, for PC- relative branches with very large displacements, BC and BALC. The 26-bit immediate, shifted left by 2 bits yields a span of 28-bits, or +/- 128 megabytes of instructions. Prior to MIPS Release 6 only the J-type instruction format (see Figure 5.8 below) provided a similar 26-bit *instr_index*.

Figure 5.5 Immediate (I-Type) Off26 CPU Instruction Format (Release 6)



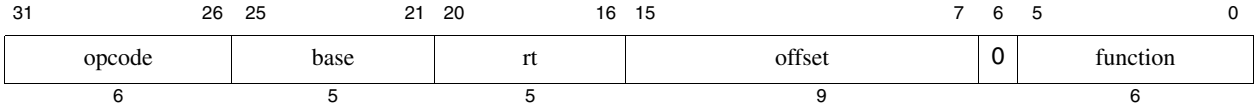
MIPS Release 6 introduces the Immediate (I-Type) Off11 CPU instruction format, shown in Figure 5.3, for new encodings of coprocessor 2 load and store instructions (LWC2, SWC2, LDC2, SWC2). This format concatenates the function and sa fields to form an 11-bit immediate.

Figure 5.6 Immediate (I-Type) Off11 CPU Instruction Format (Release 6)



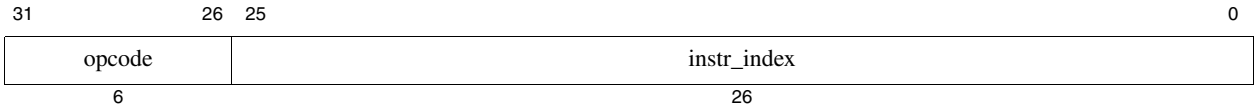
The Immediate (I-Type) Off9 CPU Instruction Format format, shown in Figure 5.3, provides a 9-bit memory offset. It was used prior to MIPS Release 6 for SPECIAL3 instructions such as EVA memory accesses (e.g., LBE). MIPS Release 6 makes extensive use of the Off9 format for instruction encodings that have been moved, such as LL and SC.

Figure 5.7 Immediate (I-Type) Off9 CPU Instruction Format (Release 6)



The Jump (J-Type) CPU instruction format is shown in Figure 5.8. Prior to Release 6 of the Architecture, instr_index was the only example of a 26-bit immediate, used in the instructions J (jump), JAL (jump-and-link), and JALX (jump and link-exchange), where the instr_index bits replaced corresponding bits in the PC. Release 6 uses the similar Off26 format for PC-relative branches.

Figure 5.8 Jump (J-Type) CPU Instruction Format



Chapter 6

FPU Programming Model

This chapter describes the MIPS32 Architecture FPU programming model. In the MIPS architecture, the FPU is implemented via Coprocessor 1, 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 includes the following sections:

6.1 Enabling the Floating Point Coprocessor

The Floating Point Coprocessor is enabled by enabling Coprocessor 1 and is a privileged operation provided by the System Control Coprocessor (CP0)). If Coprocessor 1 is not enabled, any 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.

6.2 IEEE Standard 754

IEEE Standard 754 defines the following:

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

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/.

- 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.

6.3 FPU Data Types

The FPU provides both floating-point and fixed-point data types:

- 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.

6.3.1 Floating Point Formats

The following floating point formats are provided by the FPU:

- 32-bit **single-precision** floating point (type *S*, shown in Figure 6.1)
- 64-bit **double-precision** floating point (type *D*, shown in Figure 6.2)
- 64-bit **paired-single** floating point, combining two single-precision data types (Type PS, shown in Figure 6.3)
    - The paired-single (PS) data type is removed in Release 6.

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 6.1 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

Table 6.1 Parameters of Floating Point Data Types (Continued)

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

The single and double floating-point data types are composed of three fields—*sign*, *exponent*, *fraction*—whose sizes are listed in Table 6.1.

Layouts of these fields are shown in Figure 6.1, 6.2, and 6.3 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 6.1 Single-Precision Floating Point Format (S)

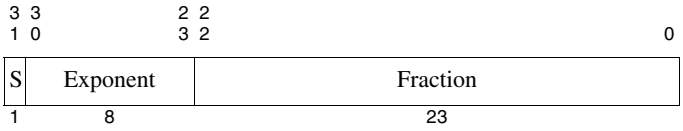


Figure 6.2 Double-Precision Floating Point Format (D)

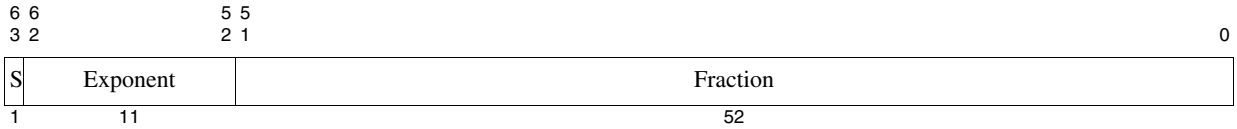
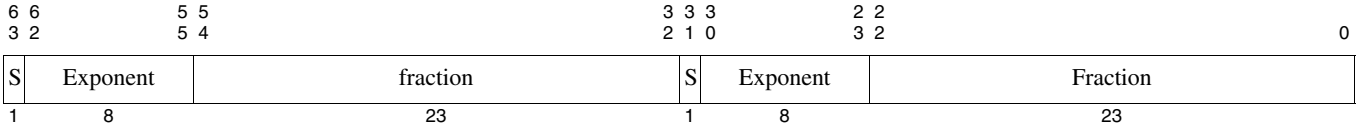


Figure 6.3 Paired-Single Floating Point Format (PS)



Values are encoded in the specified format by using unbiased exponent, fraction, and sign values listed in Table 6.2. The high-order bit of the *Fraction* field, identified as b_1, is also important for NaNs.

Table 6.2 Value of Single or Double Floating Point Data Type 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 ($FIR_{Has2008}$=0 or $FCSR_{NAN2008}$=0) Not applicable to Release 6	0x7fffffff	0x7fffffff ffffffff
			0	QNaN	Quiet NaN ($FIR_{Has2008}$=0 or $FCSR_{NAN2008}$=0) Not applicable to Release 6	0x7fbfffff	0x7ff7ffff ffffffff
E_max + 1	≠ 0		0	SNaN	Signaling NaN ($FCSR_{NAN2008}$=1)	0x7fbfffff	0x7ff7ffff ffffffff
			1	QNaN	Quiet NaN ($FCSR_{NAN2008}$=1)	0x7fffffff	0x7fffffff 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

Table 6.2 Value of Single or Double Floating Point Data Type Encoding (Continued)

Unbiased E	f	s	b_1	Value V	Type of Value	Typical Single Bit Pattern[1]	Typical Double Bit Pattern[1]
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.

6.3.1.1 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.

6.3.1.2 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 6.2, for plus infinity (+∞), minus infinity (-∞), quiet non-numbers (QNaN), and signaling non-numbers (SNaN).

6.3.1.3 Infinity and Beyond

Infinity represents a number with magnitude too large to be represented in the format and 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 (refer to the IEEE exception condition described in 6.7.2 "Exception Conditions" on page 99).

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 100).

6.3.1.4 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." In the MIPS architecture, the FPU unconditional move (MOV.fmt) and branchless conditional operations (MOVT.fmt, MOVF.fmt, MOVN.fmt, MOVZ.fmt, SEL.fmt, SELEQZ.fmt, SELNEZ.fmt) are non-arithmetic and do not signal IEEE 754 exceptions.

6.3.1.5 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 6.3 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. The $FCSR_{NAN2008}$=1 "integer QNAN" values were chosen to match the requirements of the Java and Fortran programming languages.

Table 6.3 Value Supplied When a New Quiet NaN Is Created

Format	New QNaN value ($FIR_{Has2008}$ = 0 or $FCSR_{NAN2008}$ = 0)	New QNaN value ($FCSR_{NAN2008}$ = 1)
Single floating-point	0x7fbf ffff	0x7fc0 0000
Double floating-point	0x7ff7 ffff ffff ffff	0x7ff8 0000 0000 0000
Word fixed point (result from converting any FP number too big to represent as a 32-bit positive integer)	0x7fff ffff	0x7fff ffff
Word fixed point (result from converting any FP NAN)	0x7fff ffff	0x0000 0000
Word fixed point (result from converting any FP number too small to represent as a 32-bit negative integer)	0x7fff ffff	0x8000 0000
Longword fixed point (result from converting any FP number too big to represent as a 64-bit positive integer)	0x7fff ffff ffff ffff	0x7fff ffff ffff ffff
Longword fixed point (result from converting any FP NAN)	0x7fff ffff ffff ffff	0x0000 0000 0000 0000
Longword fixed point (result from converting any FP number too small to represent as a 64-bit negative integer)	0x7fff ffff ffff ffff	0x8000 0000 0000 0000

If a CPU implements passing an input NAN operand to the output of an instruction in hardware (instead of taking an Unimplemented FP exception) and $FCSR_{NAN2008}$=1, the mantissa portion of the input NAN operand is preserved as much as possible:

- If the chosen input is a QNAN, the entire mantissa is passed to the output without change.

- If the chosen input is a SNAN, the only change is to set the leftmost/most-significant mantissa bit.

6.3.1.6 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.

Paired-single exceptions are removed in Release 6.

6.3.1.7 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**.

Paired-single condition codes are removed in Release 6.

6.3.2 Fixed Point Formats

The FPU provides two fixed-point data types:

- 32-bit **Word** fixed-point (type *W*), shown in Figure 6.4
- 64-bit **Longword** fixed-point (type *L*), shown in Figure 6.5

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 6.4 Word Fixed Point Format (W)

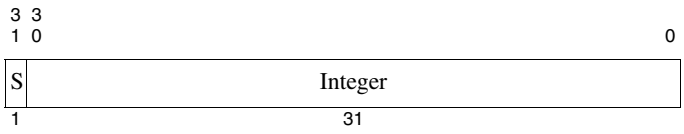
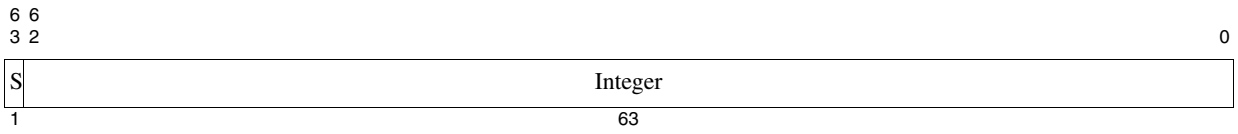


Figure 6.5 Longword Fixed Point Format (L)



6.4 Floating Point Registers

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

- *Floating Point General Purpose Registers* (*FPRs*) are 32 or 64 bits wide. These registers transfer binary data between the FPU and the system, and are also used to hold formatted FPU operand values. A 32-bit FPU contains 32, 32-bit FPRs, each of which is capable of storing a 32-bit data type. A 64-bit floating point unit contains 32, 64-bit FPRs, each of which is capable of storing any data type. The data types that each 64-bit register is capable of storing are dependent on CP0 $Status_{FR}$ (see Section 6.4.1 "FPU Register Models").

- *Floating Point Control Registers* (*FCRs*) are 32 bits wide and are used to control and provide status for all floating-point operations.

In Release 1 of the Architecture, only MIPS64 supported 64-bit floating point units with 64-bit FPRs, while MIPS32 supported only32-bit floating point units with 32-bit FPRs. In Release 2 and all subsequent releases, a 64-bit floating point unit with 64-bit FPRs is supported in both MIPS32 and MIPS64.

6.4.1 FPU Register Models

The MIPS architecture supports two FPU register models:

- *32-bit FPU register model*: 32, 32-bit registers
  - 32-bit data types stored in any register
  - pre-Release 6: 64-bit data types stored in even-odd pairs of registers.

    In Release 6 the 32-bit register model does not support 64-bit data types (stored in even-odd pairs of registers), and 64-bit operations are required to signal the Reserved Instruction exception.
- *64-bit FPU register model*: 32, 64-bit registers, with all formats supported in a register.

In Release 1 of the Architecture, MIPS32 supported only the 32-bit FPU register model (with even-odd register pairs for 64-bit data), while MIPS64 supported only the 64-bit FPU register model.

As of Release 2 and thereafter, both MIPS32 and MIPS64 support both FPU register models. If the CP0 $Status_{FR}$ bit is writable, it allows selection of the register model, whereas if this bit is read-only, it indicates which model is supported. In Release 2 and Release 3, the 32-bit FPU register model is required, while the 64-bit FPU register model is optional. In Release 5, the 64-bit FPU register model is required.

Release 6 supports both FPU register models. However, with a 64-bit FPU (FIR_{F64}=1), Release 6 requires the 64-bit FPU register model and does not support the 32-bit FPU register model, i.e., $Status_{FR}$=1 is required. With a 32-bit FPU (FIR_{F64}=0, 32-bit FPRs), Release 6 does not support 64-bit data types and requires instructions manipulating such data types to signal a Reserved Instruction exception. In particular, Release 6 does not support even-odd register pairs.

Table 6.1 below summarizes the availability and compliance requirements of FPU register widths, register models, and data types.

Table 6.4 FPU Register Models Availability and Compliance

ISA	MIPS32			MIPS64		
FPU Type	32-bit FPU FIR_{F64}=0	64-bit FPU FIR_{F64}=1		32-bit FPU FIR_{F64}=0	64-bit FPU FIR_{F64}=1	
FPU Register Width	32	64		32	64	
32-bit Data Formats S/W	32-bit data formats S and W required[1] whenever FPU is present: FIR_S=1 and FIR_W=1					
Support for 64-bit Data Types D/L	See below	FIR_D=1 and FIR_L=1		See below	FIR_D=1 and FIR_L=1	
FPU Register Model	32-bit	32-bit $Status_{FR}$=0	64-bit $Status_{FR}$=1	32-bit	32-bit $Status_{FR}$=0	64-bit $Status_{FR}$=1
64-bit Data Storage[2]	See below	**[even/odd register pairs]**	**[true 64-bit FPRs]**	See below	**[even/odd register pairs]**	**[true 64-bit FPRs]**
Release 1[3]	**[even-odd register pairs]** 64-bit data formats D/L use even/odd pairs: FIR_D=1 and FIR_L=1 required	Not Available		Not Available	Required	Required
Release 2		Required	Optional			
Release 3						
Release 5			Required			
Release 6	**[strictly 32-bit]**[4] 64-bit data formats D/L not available: FIR_D=0 FIR_L=0	Not Available		Available **[strictly 32-bit]**[4]	Not Available	

1. “Required” means “required if an FPU of specified type is present”. “Available” means that the feature is available to implement, i.e., is optional. “Not available” means that the feature cannot be implemented.
2. **[true 64-bit FPRs]** (if FIR_{F64}=1 and $Status_{FR}$=1), **[even/odd register pairs]** (if FIR_{F64}=0 or $Status_{FR}$=0, and FIR_D=FIR_L=1), and **[strictly 32-bit]** (if FIR_{F64}=FIR_D=FIR_L=0).
3. Release 1 required S, D, and W data formats, but did not require L, which was optional. Release 2, Release 3, and Release 5 required S, D, W and L data formats. Release 6 on a 64-bit FPU requires S, D, W, and L data formats; on a 32-bit FPU Release 6 requires S and W formats but 64-bit data formats D and L are not permitted.
4. The **[strictly 32-bit]** FPU mode was defined and made available as of Release 6.

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

The data transfer instructions move words and doublewords between the FPRs and the remainder of the system. The operations of the word and doubleword load and move-to instructions are shown in Figure 6.6 and Figure 6.7.

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

Figure 6.6 FPU Word Load and Move-to Operations[2]

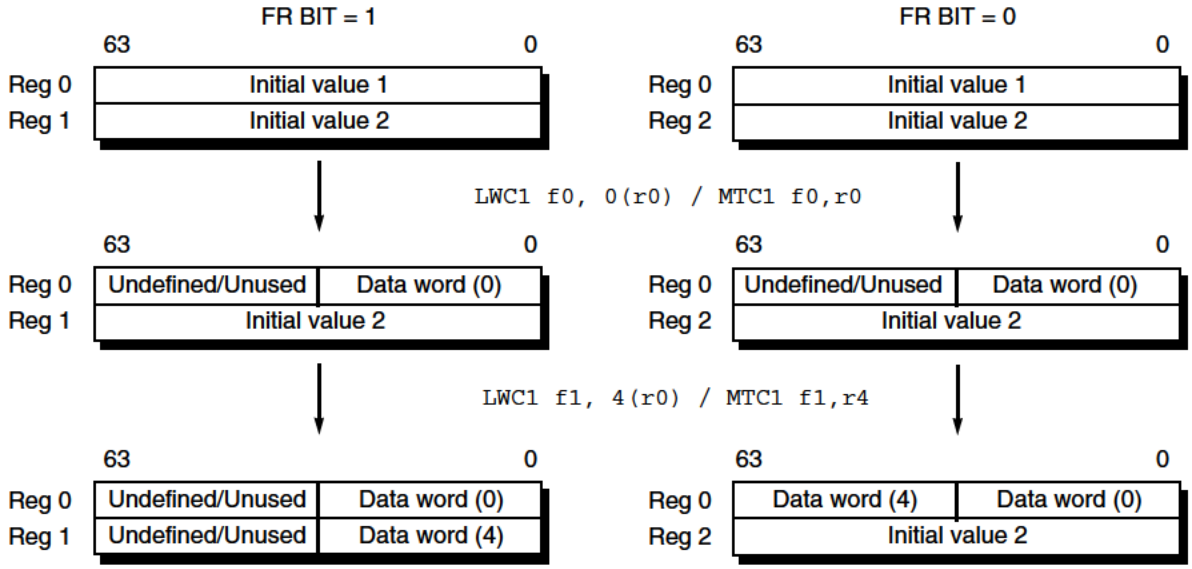
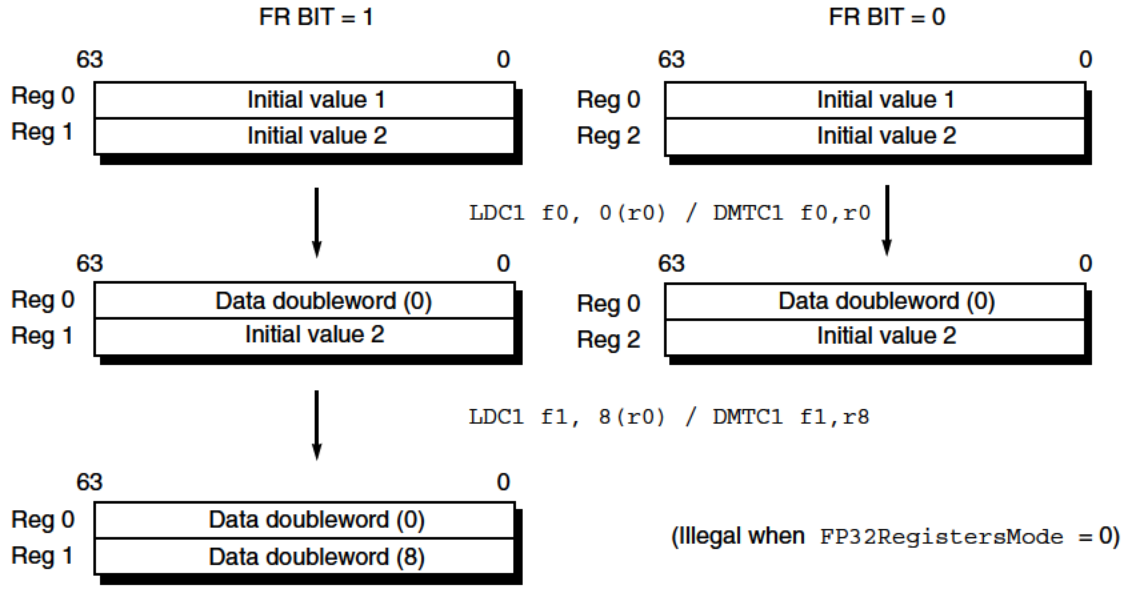


Figure 6.7 FPU Doubleword Load and Move-to Operations[3]



2. Figure 6.6 has not been updated for Release 6 (*Status*$_{FR}$=0 is not allowed with Release 6).
3. Figure 6.7 has not been updated for Release 6 (*Status*$_{FR}$=0 is not allowed with Release 6).

6.4.3 FPRs and Formatted Operand Layout

FPU instructions that operate on formatted operand values specify the FPR that contains the value. Operands that are only 32 bits wide (*W* and *S* formats), use only half of a 64-bit FPR. Operand storage of the operand types is shown in Figure 6.8, 6.9, and 6.10.

Figure 6.8 Single Floating Point or Word Fixed Point Operand in an FPR

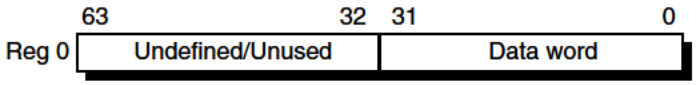


Figure 6.9 Double Floating Point or Longword Fixed Point Operand in an FPR

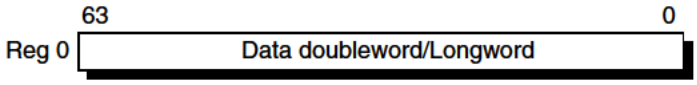
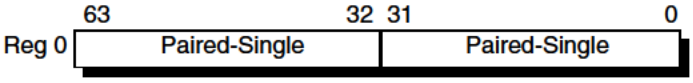


Figure 6.10 Paired-Single Floating Point Operand in an FPR (Removed in Release 6)



6.5 Floating Point Control Registers (FCRs)

The MIPS32 Architecture supports the following Floating Point Control Registers (FCRs):

- *FIR*: FP Implementation and Revision register
- *FCSR*: FP Control/Status register (used to be known as *FCR31*).
- *FEXR*: FP Exceptions register
- *FENR*: FP Enables register
- *FCCR:* FP Condition Codes register (removed in Release 6)

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. *FEXR*, *FENR*, and *FCCR* are "aliases" that allow access to the *FCSR* using CTC1 and CFC1 instructions. Release 6 removes the *FCCR*, because the FP condition codes (FCCs) have been removed.

Release 5 of the MIPS Architecture adds two additional "aliases", *UFR* and *UNFR*, for user-mode access to $Status_{FR}$. See descriptions of the CTC1 and CFC1 in Volume II for Release 5 changes to *FCSR*. These two registers are removed in Release 6, because $Status_{FR}$ has been removed.

6.5.1 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 6.11 shows the format of the *FIR* register; Table 6.5 describes the *FIR* register fields.

Figure 6.11 FIR Register Format

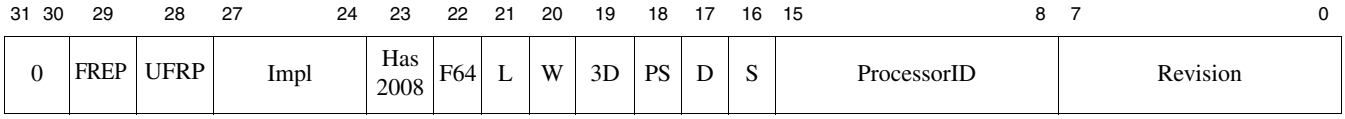


Table 6.5 FIR Register Field Descriptions

Fields		Description	Read/Write	Reset State	Compliance
Name	Bits				
0	31:30	Reserved	R	0	0
FREP	29	User-mode access of *FRE* is supported. Encoding 0: Support for emulation of $Status_{FR}$=0 handling on a 64-bit FPU with $Status_{FR}$=1 only is not available. Encoding 1: Support for emulation of $Status_{FR}$=0 handling on a 64-bit FPU with $Status_{FR}$=1 only is available. If *FREP*=1, then $Config5_{UFE}$ and $Config5_{FRE}$ are available along with CFC1/CTC1 to allow user access to *FRE*. This emulation facility is only available if an FPU is present ($Config1_{FP}$=1) and the FPU is 64-bit (FIR_{F64}=1).	R	Preset by hardware	Optional (Release 5)
UFRP	28	Indicates user-mode FR switching is supported. See Release 5 definition of CFC1 and CTC1. Encoding 0: User-mode FR switching instructions not supported. Encoding 1: User-mode FR switching instructions supported	R (Release 6) R0 (Release 6)	Preset by hardware (Release 6) 0 (Release 6)	Optional (Release 5) Reserved (Release 6)
Impl	27:24	These bits are implementation-dependent and are not defined by the architecture, other than the fact that they are read-only. These bits are explicitly not intended to be used for mode-control functions.	R	Preset	Optional
Has2008	23	Indicates that one or more IEEE-754-2008 features are implemented. If this bit is set, the *ABS2008* and *NAN2008* fields within the *FCSR* register also exist.	R	Preset by hardware (Release 5) 1 (Release 6)	Optional (Release 3) Required (Release 6)

Table 6.5 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>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 1 on any processors with a 64-bit floating point unit, and zero on any processors with a 32-bit floating point unit. A value of 1 in this bit indicates that Status_{FR} is implemented.In Release 6, Status_{FR} is read-only 1.
.
<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 by hardware</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 by hardware</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 by hardware</td><td>Required
(Release 2)</td></tr>
<tr><td>3D</td><td>19</td><td>Indicates that MIPS 3D is implemented:
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>Prior to Release 6, the MIPS 3-D ASE is optional with any 64-bit FPU. As of Release 6, it is no longer supported.</td><td>R
(pre-Release 6)

R0
(Release 6)</td><td>Preset by hardware
(pre-Release 6)

0
(Release 6)</td><td>Required
(pre-Release 6)

Reserved
(Release 6)</td></tr>
<tr><td>PS</td><td>18</td><td>Indicates 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
(pre-Release 6)

R0
(Release 6)</td><td>Preset by hardware
(pre-Release 6)

0
(Release 6)</td><td>Required
(pre-Release 6)

Reserved
(Release 6)</td></tr>
</table>

Table 6.5 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>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 by hardware</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 by hardware</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 by hardware</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 by hardware</td><td>Optional</td></tr>
</table>

6.5.2 User Floating Point Register Mode Control (UFR, CP1 Control Register 1) (Release 5 Only)

Compliance Level: *Required* in MIPS32 Release 5 if floating point is implemented and user-mode FR switching is supported. Removed by Release 6.

The *UFR* register allows user-mode to clear $Status_{FR}$ by executing a CTC1 to *UFR* with GPR[0] as input, and read $Status_{FR}$. by executing a CFC1 to UFR_{CTC1} to *UFR* with any other input register is required to produce a Reserved Instruction exception. User-mode software can determine presence of this feature from FIR_{UFRP}.

Per the definition of the CTC1 instruction, writing any value other than 0 obtained from integer GPR[0] to *UFR* using the CTC1 instruction is UNPREDICTABLE. To set UFR_{FR} / $Status_{FR}$, use CTC1 to the *UNFR FCR* alias.

Figure 6.12 UFR Register Format (pre-Release 6)

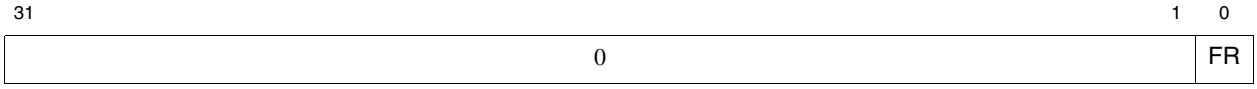


Table 6.6 UFR Register Field Descriptions (pre-Release 6)

Fields		Description	Read/ Write	Reset State	Compliance
Name	Bits				
0	31:1	Must be written as zero; returns zero on read	0	0	Reserved
FR	0	User-mode access to *Status*$_{FR}$.	R/W0[1] (Release 5) R0 (Release 6)	Undefined See reset state of *Status*$_{FR}$	Required (Release 5) R0 (Release 6)

1. *UFR* can read as 0 or 1, but can only be written with the zero from GPR[0], which clears *Status*$_{FR}$. Using CTC1 to write *UFR* with any value or GPR other than GPR[0] is UNPREDICTABLE.

6.5.3 User Negated FP Register Mode Control (UNFR, CP1 Control Register 4) (Removed in Release 6)

Compliance Level: *Required* in MIPS32 Release 5 if floating point is implemented and user-mode FR switching is supported. Removed in Release 6.

The *UNFR* register allows user-mode to set *Status*$_{FR}$ by executing a CTC1 to *UNFR* with GPR[0] as input. CTC1 to *UNFR* with any other input register is required to produce a Reserved Instruction exception. User-mode software can determine presence of this feature from *FIR*$_{UFRP}$.

Figure 6.13 UNFR Register Format (pre-Release 6)

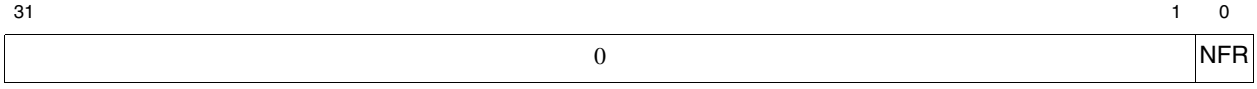


Table 6.7 UNFR Register Field Descriptions (pre-Release 6)

Fields		Description	Read/ Write	Reset State	Compliance
Name	Bits				
0	31:1	Must be written as zero.	0	Undefined	Reserved
NFR	0	User-mode inverted write to *Status*$_{FR}$.	W0[1] R0 (Release 6)	Undefined	Required (Release 5) R0 (Release 6)

1. UNFR can only be written with the zero from GPR[0], which sets *Status*$_{FR}$. Using CFC1 to read UNFR, or using CTC1 to write UNFR with any value or GPR other than GPR[0] is UNPREDICTABLE. UNFR's "state" can be inferred by reading *Status*$_{FR}$, e.g., via UFR.

The *UNFR* pseudo-control-register alias is a convenience, allowing CTC1 $0, *UNFR* to be used to set *UFR*/ *Status*$_{FR}$ without requiring a GPR to hold the value such as 1 to be written. Because reading *UNFR* would be redun-

dant with reading *UFR*, *UNFR* is write-only; attempting to read *UNFR* via CFC1 is UNPREDICTABLE, per the definition of the CFC1 instruction. Writing any value other than 0 obtained from integer GPR $0 to *UNFR* using the CTC1 instruction is similarly UNPREDICTABLE from software's point of view, and is required to produce a Reserved Instruction exception in Release 5 implementations.

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

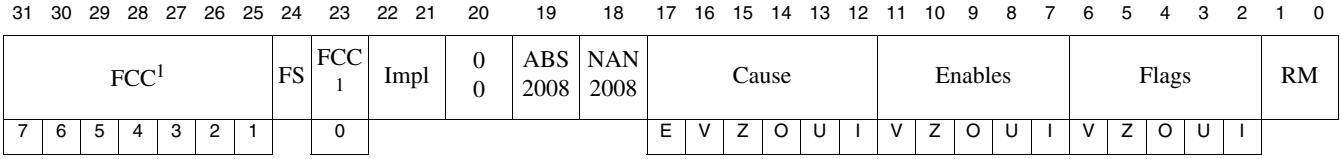
Compliance Level: *Required* if floating point is implemented.

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
- pre-Release 6: indicates the condition code result of FP compare instructions
- Release 6 removes the FP condition codes.

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 6.14 shows the format of the *FCSR* register; Table 6.8 describes the *FCSR* register fields.

Figure 6.14 FCSR Register Format



1. Release 6 removes the FCCs.

Table 6.8 FCSR Register Field Descriptions

Fields		Description	Read/ Write	Reset State	Compliance
Name	Bits				
FCC (pre-Release 6)	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
FCC (Release 6)	31:25, 23	Floating point condition codes. pre-Release 6 feature, removed by Release 6.	R0	0	Reserved

Table 6.8 FCSR 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>FS</td><td>24</td><td>Flush to Zero (Flush Subnormals).
See sections 6.7.2.3 “Underflow Exception” on page 101 and 6.7.2.4 “Alternate Flush to Zero Underflow Handling” on page 101.

<table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>Input subnormal values and tiny non-zero results are not altered. Unimplemented Operation exception may be signaled as needed.</td></tr><tr><td>1</td><td>When FS is one, subnormal results are flushed to zero. The Unimplemented Operation exception is NOT signalled for this reason.

Every tiny non-zero result is replaced with zero of the same sign.

Prior to Release 5, it is implementation- dependent whether subnormal operand values are flushed to zero before the operation is carried out.

As of Release 5, every input subnormal value is replaced with zero of the same sign.</td></tr></table></td><td>R/W</td><td>Undefined</td><td>Required</td></tr>
<tr><td>Impl</td><td>22:21</td><td>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.</td><td>R/W</td><td>Undefined</td><td>Optional</td></tr>
<tr><td>0</td><td>20</td><td>Reserved for future use; reads as zero.</td><td>R</td><td>Preset by hardware</td><td>Reserved</td></tr>
<tr><td>ABS2008</td><td>19</td><td>ABS.fmt and NEG.fmt instructions compliant with IEEE Standard 754-2008.
The IEEE 754-2008 standard requires that the ABS and NEG functions are non-arithmetic and accept NAN inputs without trapping.

This fields exists if $FIR_{Has2008}$ is set.

<table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>ABS and NEG instructions are arithmetic and trap for NAN input. MIPS legacy behavior.</td></tr><tr><td>1</td><td>ABS and NEG instructions are non-arithmetic and accept NAN input without trapping. IEEE 754-2008 behavior</td></tr></table></td><td>R</td><td>Preset by hardware (pre-Release 6)

1 (Release 6)</td><td>Optional as of Release 3.50

Required as of Release 5</td></tr>
</table>

Table 6.8 FCSR Register Field Descriptions (Continued)

Fields		Description	Read/ Write	Reset State	Compliance
Name	Bits				
NAN2008	18	Quiet and signaling NaN encodings recommended by the IEEE Standard 754-2008, i.e., a quiet NaN is encoded with the first bit of the fraction being 1 and a signaling NaN is encoded with the first bit of the fraction field being 0. MIPS legacy FPU encodes NaN values with the opposite polarity, i.e., a quiet NaN is encoded with the first bit of the fraction being 0 and a signaling NaN is encoded with the first bit of the fraction field being 1. Refer to Table 6.3 for the quiet NaN encoding values. This fields exists if $FIR_{Has2008}$ is set. Encoding: 0 — MIPS legacy NaN encoding; 1 — IEEE 754-2008 NaN encoding	R	Preset by hardware 1 (Release 6)	Optional as of Release 3.50 Required as of Release 5
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 6.9 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 *Enables* 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 *Enables* bit; the non-IEEE Unimplemented Operation exception is defined by MIPS as always enabled. Refer to Table 6.9 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 *Enables* bit was off), the corresponding bit(s) in the *Flags* field are set, while the others remain unchanged. Arithmetic operations that result in a Floating Point exception (i.e., the *Enables* 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 6.9 for the meaning of each bit.	R/W	Undefined	Required

Table 6.8 FCSR Register Field Descriptions (Continued)

Fields		Description	Read/ Write	Reset State	Compliance
Name	Bits				
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 6.10 for the meaning of the encodings of this field.	R/W	Undefined	Required

Several registers are “aliases” for particular fields in *FCSR* in order to make it easier to modify them in a single write rather than having to perform a read modify write. These FP control register aliases include:

- *FEXR* - access to Cause and Flags fields of *FCSR*
- *FENR* - access to Enables, FS, and RM fields of *FCSR*
- pre-Release 6: *FCCR* - access to the Floating Point Condition Codes (FCCs) of *FCSR*
    - Release 6 removes the FCCs and *FCCR*.

The aliased fields in the *FEXR*, *FENR*, pre-Release 6 *FCCR*, and *FCSR* registers always display the correct state. That is, if a field is written via *FCSR*, 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 6.9 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 6.10 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.

6.5.5 Floating Point Condition Codes Register (FCCR, CP1 Control Register 25) (pre-Release 6)

Compliance Level: pre-Release 6: *Required* if floating point is implemented.
Release 6 removes the FCCs and *FCCR*: this register is reserved in Release 6.

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 6.15 shows the format of the *FCCR* register; Table 6.11 describes the *FCCR* register fields.

Figure 6.15 FCCR Register Format

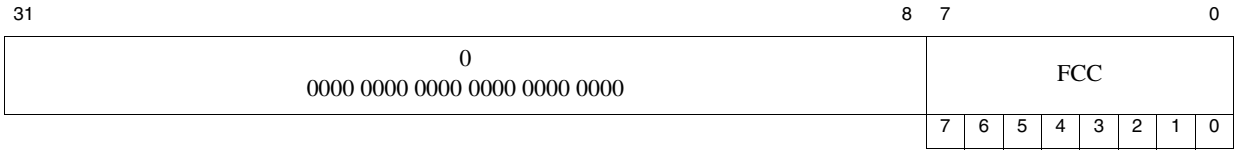


Table 6.11 FCCR Register Field Descriptions

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 (pre-Release 6) R0 (Release 6)	Undefined (pre-Release 6) 0 (Release 6)	Required

6.5.6 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 6.16 shows the format of the *FEXR* register; Table 6.12 describes the *FEXR* register fields.

Figure 6.16 FEXR Register Format

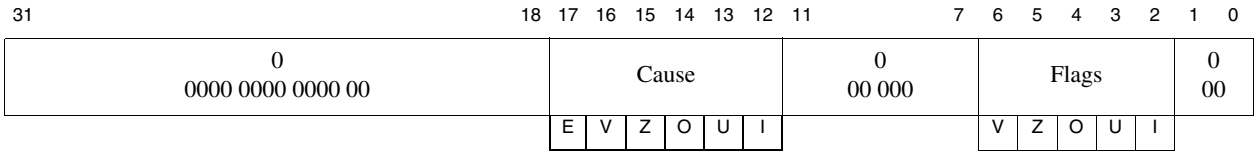


Table 6.12 FEXR Register Field Descriptions

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

6.5.7 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 6.17 shows the format of the *FENR* register; Table 6.13 describes the *FENR* register fields.

Figure 6.17 FENR Register Format

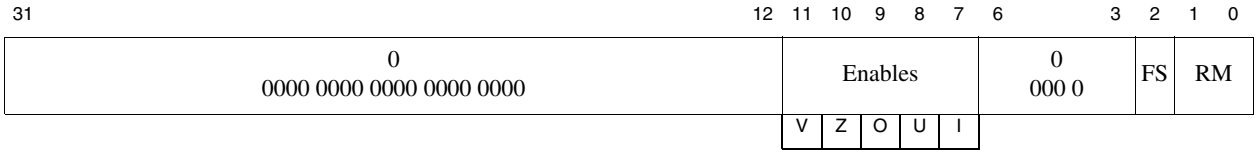


Table 6.13 FENR Register Field Descriptions

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

6.6 Formats and Sizes of Floating Point Data

6.6.1 Formats of Values Used in FP Registers

Unlike the CPU, the FPU does not interpret the binary encoding of source operands nor produce a binary encoding of results for every operation. The value held in a floating point register (FPR) is either *uninterpreted*, or one of the valid numeric formats: *single*, *double*, *paired-single* floating-point, *word* and *long* fixed point.

The value in a FPR is set to one of these formats when the register is written:

- When a data transfer instruction writes binary data into a FPR (LWC1, LDC1, MTC1, MTHC1, pre-Release 6: LWXC1, LDXC1, LUXC1), then the binary value of the register is *uninterpreted*.
- A FP computational or FP register move (MOV*.fmt) instruction which produces a result of type *fmt* puts a value of type *fmt* into the result register.
- The format of the value of a FPR is unchanged when it is read by data transfer instruction (SWC1, SDC1, MFC1, MFHC1, pre-Release 6: SWXC1, SDXC1, SUXC1).

When an FPR with an uninterpreted value is used as a source operand by an instruction that requires a value of format *fmt*, the binary contents are interpreted as a value of format *fmt*. A FP arithmetic instruction produces a value of the expected numeric format into the destination register.

If an FPR contains a value of numeric format *fmt* and an instruction uses the FPR as source operand of different numeric format, the result of the instruction is **UNPREDICTABLE**.

6.6.2 Sizes of Floating Point Data

Any FPU instruction that writes a 32-bit result (S or W format) to the low 32-bits of a 64-bit FPR leaves the upper bits of the FPR **UNPREDICTABLE**. This includes the 32-bit FPU 32-bit load LWC1, the data transfer instruction MTC1, as well as any instruction that computes a 32 bit result, e.g., ADD.S, CMP.condn.S, CLASS.S.

The MIPS SIMD Architecture (MSA) is an architecture module that operates on 128-bit vector registers, which are overlaid upon the FPU register file: FPR[0] occupies the low 32-bits or 64-bits of the 128-bit MSA vector register W[0], and so on. Any FPU instruction that writes an FPR writes a 32-bit or 64-bit value, and leaves the upper bits of the corresponding MSA register **UNPREDICTABLE**, i.e., a 32-bit FPU instruction leaves bits 32-127 of the MSA register unpredictable, while a 64-bit FPU instruction leaves bits 64-127 **UNPREDICTABLE**.

6.7 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.

6.7.1 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 6.14). 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.

6.7.2 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 6.14.

Table 6.14 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.

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

6.7.2.2 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.

6.7.2.3 Underflow Exception

This section describes IEEE standard compliant underflow exception handling, desired when *FCSR*$_{FS}$=0. Some implementations may require software assistance to accomplish this, via the Unimplemented Operation exception handler. See the next section, 6.7.2.4, for Alternate Flush to Zero Underflow Handling, obtained by setting *FCSR*$_{FS}$=1, which may be faster on some implementations.

Two related events contribute to underflow:

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

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

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

The MIPS architecture specifies that tininess be detected after rounding.

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

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

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

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

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

6.7.2.4 Alternate Flush to Zero Underflow Handling

Previous section 6.7.2.3 "Underflow Exception" describes IEEE standard compliant underflow exception handling, desired when *FCSR*$_{FS}$=0. The current section describes Alternate Flush to Zero Underflow Handling, obtained by setting *FCSR*$_{FS}$=1,which never requires the Unimplemented Operation exception handler to handle subnormal results, and which may be faster on some implementations even if software exception handler assistance is not required.

When the $FCSR_{FS}$ is set:

Results: Every tiny non-zero result is replaced with zero of the same sign.

Inputs: Prior to Release 5, it is implementation-dependent whether subnormal operand values are flushed to zero before the operation is carried out. As of Release 5, every input subnormal value is replaced with zero of the same sign.

Exceptions: Because the $FCSR_{FS}$ bit flushes subnormal results to zero, the Unimplemented Operation exception will never be produced for this reason. All the other floating point exceptions are signaled according to the new values of the operands or the results. In addition, when the $FCSR_{FS}$ bit is set:

- Tiny non-zero results are detected before rounding[4]. Flushing of tiny non-zero results causes Inexact and Underflow exceptions to be signaled.
- Flushing of subnormal input operands in all instructions except comparisons causes Inexact exception to be signaled.
- For floating-point comparisons, the Inexact exception is not signaled when subnormal input operands are flushed.
- Inputs to non-arithmetic floating-point instructions are never flushed.

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

6.7.2.5 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.

6.7.2.6 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

6.7.2.7 Unimplemented Operation Exception

The Unimplemented Operation exception is a MIPS-defined exception that provides support for software emulation. 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.

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

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.

Note: the Unimplemented Operation may be signalled when the FPU state involved exists, but where hardware cannot complete the operation. In particular, the Unimplemented Operation may be signalled when hardware can complete the operation for some, but not all, inputs. The Unimplemented Operation is not signalled when hardware does not implement the FPU state involved in the operation[5]. For example, the Unimplemented Operation is not signalled when a 32-bit SP-only FPU executes a 64-bit instruction, FIR_S=FIR_W=1, FIR_D=FIR_L=0. Instead, a Reserved Instruction exception is signalled.

5. With the possible exception of minor control bits.

Chapter 7

FPU Instruction Set

This chapter describes the MIPS32instruction set architecture for the floating point unit (FPU) in the following sections:

7.1 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 backward-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 of the Architecture and subsequent releases, a 32-bit CPU may include a full 64-bit coprocessor, including a floating point unit that implements the same mode bit to select a 32-bit or 64-bit FPU register model. As of Release 5 of the Architecture, if floating point is implemented, then FR=1 is required, i.e., the 64-bit FPU, with the FR=1 64-bit FPU register model, is required. As of Release 6, 64-bit FPUs (FIR_{F64}=1) no longer support the 32-bit FPU register model, that is, $Status_{FR}$=0 is not supported, and $Status_{FR}$=1 is required and cannot be changed; whereas on a 32-bit FPU (FIR_{F64}=0), 64-bit data formats D and L cannot be supported (FIR_D=FIR_L=0).

Release 6 does not provide the even-odd register pair implementation of 64-bit double-precision on a 32-bit register file. Release 6 provides only the flat, unpaired, 64-bit register model (as if $Status_{FR}$=1); or flat, unpaired, 32-bit floating point registers, on a 32-bit FPU that implements 32-bit single-precision (S) and word (W) operations. but not 64-bit double-precision (D) or longword (L) operations, which trap (and can therefore be emulated).

In all architectures prior to Release 6, any processor implementing MIPS64 can also run MIPS32 binary programs built for the same, or a lower release of the Architecture, without change. For Release 6, any processor implementing MIPS64 Release 6 can also run MIPS32 Release 6 binary programs without change, so long as the same instruction-set subsets are provided.

7.2 FPU Instructions

The FPU instructions comprise the following functional groups:

- Arithmetic Instructions
- Conversion Instructions
- Formatted Operand-Value Move Instructions
- FPU Conditional Branch Instructions.
- Miscellaneous Instructions (Removed in Release 6)

FPU instructions are listed and described by type in Tables 7.2 through 7.19Table 7.2 Table 7.19. The instruction tables specify the MIPS architecture ISA(s) in which the instruction is defined. "MIPS32" indicates that the operation is present in all revisions of MIPS32, "MIPS64, MIPS32 Release 2" indicates that the operation is present in all versions of MIPS64 and is present in MIPS32 Release 2 and all later versions of MIPS32, unless otherwise noted; "Release 6" indicates that the operation is present in Release 6 and not in previous revisions; "Removed in Release 6" means that implementations of Release 6 are required to signal the Reserved Instruction exception when there is no higher priority exception, and when the instruction encoding has not been reused for a different instruction

7.2.1 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 7.1.

Table 7.1 FPU Data Transfer Instructions

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

7.2.1.1 Data Alignment in Loads, Stores, and Moves

pre-Release 6: 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.

Release 6 requires that a Release 6 compliant system support misaligned memory accesses (possibly by trapping and emulating) for all ordinary memory access instructions, including coprocessor loads and stores. Nevertheless, it is recommended that software naturally align FPU data in memory, for improved performance.

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 43).

7.2.1.2 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.

Release 6 removes loads and stores that have the Pre-Release 6 *register+register* addressing mode. Release 6 provides instructions such as LSA (Load Scaled Address) that perform the *register+register*scale* computation commonly involved in accessing data structures such as arrays, separate from the memory access.

Tables 7.2 through 7.4 list the FPU data transfer instructions.

Table 7.2 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 7.3 FPU Loads and Using Register+Register Address Mode (Removed in Release 6)

Mnemonic	Instruction	Defined in MIPS ISA
LDXC1	Load Doubleword Indexed to Floating Point	MIPS64 MIPS32 Release 2 Removed in Release 6
LUXC1	Load Doubleword Indexed Unaligned to Floating Point	MIPS64 MIPS32 Release 2 Removed in Release 6
LWXC1	Load Word Indexed to Floating Point	MIPS64 MIPS32 Release 2 Removed in Release 6
SDXC1	Store Doubleword Indexed to Floating Point	MIPS64 MIPS32 Release 2 Removed in Release 6
SUXC1	Store Doubleword Indexed Unaligned to Floating Point	MIPS64 MIPS32 Release 2 Removed in Release 6
SWXC1	Store Word Indexed to Floating Point	MIPS64 MIPS32 Release 2 Removed in Release 6

Table 7.4 FPU Move To and From Instructions

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

7.2.2 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).

7.2.2.1 FPU IEEE Arithmetic Instructions

FPU IEEE-approximate arithmetic operations are listed in Table 7.5.

Table 7.5 FPU IEEE Arithmetic Operations

Mnemonic	Instruction	Defined in MIPS ISA
ABS.fmt	Floating Point Absolute Value (Arithmetic if $FIR_{Has2008}$=0 or $FCSR_{ABS2008}$=0)	MIPS32
ABS.fmt (*PS*)	Floating Point Absolute Value (Paired Single) (Arithmetic if $FIR_{Has2008}$=0 or $FCSR_{ABS2008}$=0)	MIPS64 MIPS32 Release 2 Removed in Release 6
ADD.fmt	Floating Point Add	MIPS32
ADD.fmt (*PS*)	Floating Point Add (Paired Single)	MIPS64 MIPS32 Release 2 Removed in Release 6
C.cond.fmt	Floating Point Compare (setting FCC)	MIPS32 Removed in Release 6
C.cond.fmt (*PS*)	Floating Point Compare (Paired Single) (setting FCC)	MIPS64 MIPS32 Release 2 Removed in Release 6
CMP.cond.fmt	Floating Point Compare (setting FPR)	Release 6
DIV.fmt	Floating Point Divide	MIPS32
MUL.fmt	Floating Point Multiply	MIPS32
MUL.fmt (*PS*)	Floating Point Multiply (Paired Single)	MIPS64 MIPS32 Release 2 Removed in Release 6
NEG.fmt	Floating Point Negate (Arithmetic if $FIR_{Has2008}$=0 or $FCSR_{ABS2008}$=0)	MIPS32
NEG.fmt (*PS*)	Floating Point Negate (Paired Single) (Arithmetic if $FIR_{Has2008}$=0 or $FCSR_{ABS2008}$=0)	MIPS64 MIPS32 Release 2 Removed in Release 6
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 Removed in Release 6

7.2.2.2 FPU non-IEEE-approximate Arithmetic Instructions

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 7.6.

Table 7.6 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

7.2.2.3 FPU Multiply-add Instructions

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 7.7.

Release 6 removes these instructions and replaces them by fused multiply-add instructions, listed in Table 7.8.

Arithmetic and rounding behavior: 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 7.7 lists the FPU non-fused Multiply-Accumulate arithmetic operations.

Table 7.7 FPU Multiply-Accumulate Arithmetic Operations (Removed in Release 6)

Mnemonic	Instruction	Defined in MIPS ISA
MADD.fmt	Floating Point Multiply Add	MIPS64 MIPS32 Release 2 Removed in Release 6
MADD.fmt (*PS*)	Floating Point Multiply Add (Paired Single)	MIPS64 MIPS32 Release 2 Removed in Release 6
MSUB.fmt	Floating Point Multiply Subtract	MIPS64 MIPS32 Release 2 Removed in Release 6
MSUB.fmt (*PS*)	Floating Point Multiply Subtract (Paired Single)	MIPS64 MIPS32 Release 2 Removed in Release 6
NMADD.fmt	Floating Point Negative Multiply Add	MIPS64 MIPS32 Release 2 Removed in Release 6
NMADD.fmt (*PS*)	Floating Point Negative Multiply Add (Paired Single)	MIPS64 MIPS32 Release 2 Removed in Release 6

Table 7.7 FPU Multiply-Accumulate Arithmetic Operations (Removed in Release 6) (Continued)

Mnemonic	Instruction	Defined in MIPS ISA
NMSUB.fmt	Floating Point Negative Multiply Subtract	MIPS64 MIPS32 Release 2 Removed in Release 6
NMSUB.fmt (*PS*)	Floating Point Negative Multiply Subtract (Paired Single)	MIPS64 MIPS32 Release 2 Removed in Release 6

7.2.2.4 FPU Fused Multiply-Accumulate instructions (Release 6)

Release 6 provides IEEE 2008 compliant fused multiply-accumulate add (MADDF.fmt) and subtract (MSUBF.fmt) instructions. These instructions are listed in Table 7.8.

Arithmetic and rounding behavior: The product is calculated to mimic infinite precision. The accumulated result is rounded according to the current rounding mode. This model meets the IEEE-754-2008 specification. This behavior is often known as “Fused”.

Table 7.8 lists the FPU Fused Multiply-Accumulate arithmetic operations.

Table 7.8 FPU Fused Multiply-Accumulate Arithmetic Operations (Release 6)

Mnemonic	Instruction	Defined in MIPS ISA
MADDF.fmt	Fused Floating Point Multiply Add	MIPS32 Release 6
MSUBF.fmt	Fused Floating Point Multiply Subtract	MIPS32 Release 6

7.2.2.5 Floating Point Comparison Instructions

Floating point comparison instructions are listed in Table 7.9.

Prior to Release 6, the C.cond.fmt instruction compares two floating point values, writing the condition to one of the floating point condition codes. Release 6 removes the FCCs and related instructions.

Release 6 adds the CMP.cond.fmt instruction that compares two floating point values and writes a bit mask of all 0s or all 1s the width of the fmt specified. Bits beyond the format are UNPREDICTABLE. This mask can be used in logical operations such as MSA vector operations, or the least-significant bit, bit 0, can be tested by FPU branches such as BC1EQZ/BC1NEZ, or by floating point select instructions SEL.fmt, SELEQZ.fmt, SELNEZ.fmt. This instruction adds several comparisons that the C.cond.fmt instruction does not provide. It can produce both polarities for all comparisons.

Release 6 adds the CLASS.fmt instruction, which corresponds to the class operation of the IEEE Standard for Floating-Point Arithmetic 754TM-2008. It produces a bitmask whose set bits indicate important properties of its argument: SNaN, QNaN, +/- infinity, normal, subnormal, zero. CLASS.fmt does not cause an exception if its operand is a NaN, not even for an SNaN.

Release 6 adds the MAX/MIN/MAXA/MINA.fmt family of instructions that correspond to IEEE 2008 **maxNum**, **minNum**, **maxNumArg**, and **MinNumArg**. They are NaN suppressing, returning a numeric value even if one of the arguments is a NaN. MAXA and MINA return the value that has the largest magnitude without changing the sign.

Table 7.9 Floating Point Comparison Instructions

Mnemonic	Instruction	Defined in MIPS ISA
C.cond.fmt	Floating Point Compare (setting FCC)	MIPS32 Removed in Release 6
C.cond.fmt (*PS*)	Floating Point Compare (Paired Single) (setting FCC)	MIPS64 MIPS32 Release 2 Removed in Release 6
CLASS.fmt	Scalar Floating-Point Class Mask	MIPS32 Release 6
CMP.cond.fmt	Floating Point Compare (setting FPR)	MIPS32 Release 6
MAX.fmt	Floating Point Maximum	MIPS32 Release 6
MAXA.fmt	Floating Point Value with Maximum Absolute Value	MIPS32 Release 6
MIN.fmt	Floating Point Minimum	MIPS32 Release 6
MINA.fmt	Floating Point Value with Minimum Absolute Value	MIPS32 Release 6

7.2.3 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.

CVT.W.fmt and CVT.L.fmt convert from fmt to W and L respectively - 2's complement binary integers stored in the floating point registers.

Release 6 adds the RINT.fmt instruction that changes a floating-point value to an integer—the result is an integer value, but represented in floating-point format.

Tables 7.10 and 7.11 list the FPU conversion instructions according to their rounding mode.

Table 7.10 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 Removed in Release 6
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 Removed in Release 6
CVT.W.fmt	Floating Point Convert to Word Fixed Point	MIPS32
RINT.fmt	Scalar floating-point round to integer	MIPS32 Release 6

Table 7.11 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
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

7.2.4 Formatted Operand-Value Move Instructions

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

- Unconditional move
- Instructions which modify the sign bit (ABS.fmt and NEG.fmt when $FCSR_{ABS2008}$=1)
- FPU conditional move instructions (removed in Release 6)
  - Conditional move that tests an FPU true/false condition code
  - Conditional move that tests a CPU general-purpose register against zero
- FPU conditional select instructions, based on testing bit 0 of an FPT (Release 6)
  - Full conditional select between two FPRs (SEL.fmt)
  - Conditional select that selects between an FPR and zero (SELRQZ.fmt, SELNEZ.fmt)

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.)

The FPU unconditional move, absolute value, and negate instructions are listed in Table 7.12.

Table 7.12 FPU Formatted Unconditional Operand Move Instructions

Mnemonic	Instruction	Defined in MIPS ISA
ABS.fmt	Floating Point Absolute Value (Non-Arithmetic if $FCSR_{ABS2008}$=1)	MIPS32
ABS.fmt (*PS*)	Floating Point Absolute Value (Paired Single) (Non-Arithmetic if $FCSR_{ABS2008}$=1)	MIPS64 MIPS32 Release 2 Removed in Release 6
MOV.fmt	Floating Point Move	MIPS32
MOV.fmt (*PS*)	Floating Point Move (Paired Single)	MIPS64 MIPS32 Release 2 Removed in Release 6
NEG.fmt	Floating Point Negate (Non-Arithmetic if $FCSR_{ABS2008}$=1)	MIPS32
NEG.fmt (*PS*)	Floating Point Negate (Paired Single) (Non-Arithmetic if $FCSR_{ABS2008}$=1)	MIPS64 MIPS32 Release 2 Removed in Release 6

The FPU conditional move instructions are listed in Table 7.13 andTable 7.14

Table 7.13 FPU Conditional Move on True/False Instructions (Removed in Release 6)

Mnemonic	Instruction	Defined in MIPS ISA
MOVF.fmt	Floating Point Move Conditional on FP False	MIPS32 Removed in Release 6
MOVF.fmt (*PS*)	Floating Point Move Conditional on FP False (Paired Single)	MIPS64 MIPS32 Release 2 Removed in Release 6
MOVT.fmt	Floating Point Move Conditional on FP True	MIPS32 Removed in Release 6
MOVT.fmt (*PS*)	Floating Point Move Conditional on FP True (Paired Single)	MIPS64 MIPS32 Release 2 Removed in Release 6

Table 7.14 FPU Conditional Move on Zero/Nonzero Instructions (Removed in Release 6)

Mnemonic	Instruction	Defined in MIPS ISA
MOVN.fmt	Floating Point Move Conditional on Nonzero	MIPS32 Removed in Release 6
MOVN.fmt (PS)	Floating Point Move Conditional on Nonzero (Paired Single)	MIPS64 MIPS32 Release 2 Removed in Release 6
MOVZ.fmt	Floating Point Move Conditional on Zero	MIPS32 Removed in Release 6
MOVZ.fmt (PS)	Floating Point Move Conditional on Zero (Paired Single)	MIPS64 MIPS32 Release 2 Removed in Release 6

The Release 6 FPU conditional select instructions are listed in Table 7.15

Table 7.15 FPU Conditional Select Instructions (Release 6)

Mnemonic	Instruction	Defined in MIPS ISA
SEL.fmt	Floating Point Select	MIPS32 Release 6
SELEQZ.fmt	Floating Point Select if condition Equal to Zero, Else 0.0	MIPS32 Release 6
SELNEZ.fmt	Floating Point Select if condition is Not Equal to Zero, Else 0.0	MIPS32 Release 6

7.2.5 FPU Conditional Branch Instructions.

In releases prior to Release 6, the FPU has PC-relative conditional branch instructions that test condition codes set by FPU compare instructions (C.cond.fmt), and eight condition codes are defined for use in compare and branch instructions. For backward compatibility with previous revisions of the ISA, condition code bit 0 and condition code bits 1 through 7 are in discontiguous fields in *FCSR*.

Release 6 removes the FPU condition codes and related instructions, such as C.cond.fmt.

Release 6 provides PC-relative conditional branch operations that test the least significant bit, bit 0, of an FPR (BC1EQZ, BC1NEZ). This FPU truth value is produced by a Release 6 compare instruction (CMP.cond.fmt) that sets a mask of all 0s or all 1s in an FPR destination register, the width of the operands; or by similar integer compare operations.

All branches (before the introduction of compact branches in microMIPS and Release 6) 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.

In releases prior to Release 6, 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.
- pre-Release 6: **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). Branch likely instructions are removed in Release 6.

Table 7.16 lists the FPU conditional branch instructions; Table 7.17 lists the deprecated conditional Branch likely instructions.

Table 7.16 FPU Conditional Branch Instructions (Removed in Release6)

Mnemonic	Instruction	Defined in MIPS ISA
BC1F	Branch on FP False	MIPS32 Removed in Release 6
BC1T	Branch on FP True	MIPS32 Removed in Release 6

Table 7.17 Deprecated FPU Conditional Branch Likely Instructions (Removed in Release 6)

Mnemonic	Instruction	Defined in MIPS ISA
BC1FL	Branch on FP False Likely	MIPS32 Removed in Release 6
BC1TL	Branch on FP True Likely	MIPS32 Removed in Release 6

Table 7.18 lists the Release 6 FPU conditional branch instructions, which branch according to an FP condition in bit 0 of an operand register.

Table 7.18 FPU Conditional Branch Instructions (Release 6)

Mnemonic	Instruction	Defined in MIPS ISA
BC1EQZ	Branch on FP condition Equal to Zero	MIPS32 Release 6
BC1NEZ	Branch on FP condition Not Equal to Zero	MIPS32 Release 6

7.2.6 Miscellaneous Instructions (Removed in Release 6)

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). All these instructions are removed in Release 6.

Table 7.19 lists the conditional move instructions.

Table 7.19 Miscellaneous Instructions (Removed in Release 6)

Mnemonic	Instruction	Defined in MIPS ISA
ALNV.PS	FP Align Variable	MIPS64 MIPS32 Release 2 Removed in Release 6
MOVN.fmt	Move Conditional on FP False	MIPS32 Removed in Release 6
MOVZ.fmt	Move Conditional on FP True	MIPS32 Removed in Release 6
PLL.PS	Pair Lower Lower	MIPS64 MIPS32 Release 2 Removed in Release 6
PLU.PS	Pair Lower Upper	MIPS64 MIPS32 Release 2 Removed in Release 6
PUL.PS	Pair Upper Lower	MIPS64 MIPS32 Release 2 Removed in Release 6
PUU.PS	Pair Upper Upper	MIPS64 MIPS32 Release 2 Removed in Release 6

7.3 Valid Operands for FPU Instructions

The FPU 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 7.20.

Table 7.20 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[1]	paired single	64 (2x32)	Floating point
23–31	7	Reserved			

1. PS is removed in Release 6

In Release 6, certain instructions (CMP.condn.fmt) use the W and L integer formats to indicate the S and D floating-point formats, respectively. Release 6 removes the PS format.

The result of an instruction using operand formats marked **U** in Table 7.21 is not currently specified by the architecture and causes a Reserved Instruction exception.

Table 7.21 Valid Formats for FPU Operations

Mnemonic	Operation	Operand Fmt					COP1 Function Value	COP1X op4 Value
		Float			Fixed			
		S	D	PS[1]	W	L		
ABS	Absolute value	•	•	•	**U**	**U**	5	
ADD	Add	•	•	•	**U**	**U**	0	
C.*cond*	Floating Point Compare (FCC)	•	•	•	**U**	**U**	48–63	
CLASS	Classify FP Value	•	•				27	
CMP.*cond*	Floating Point Compare (FPR condition)	•	•	**U**	**S**	**D**	48–63	
		CMP.cond uses the W and L to indicate S and D, respectively.						
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	

Table 7.21 Valid Formats for FPU Operations (Continued)

Mnemonic	Operation	Operand Fmt					COP1 Function Value	COP1X op4 Value
		Float			Fixed			
		S	D	PS[1]	W	L		
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 $-\infty$	•	•	**U**	**U**	**U**	11 (15)	
MADD	Multiply-Add	•	•	•	**U**	**U**		4
MADDF	Fused Multiply-Add	•	•				24	
MAX	Maximum	•	•				29	
MAXA	Maximum Magnitude Argument	•	•				31	
MIN	Minimum	•	•				28	
MINA	Minimum Magnitude Argument	•	•				30	
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
MSUBF	Fused Multiply-Subtract	•	•				25	
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	
RINT	Round to Integer Floating Point Value	•	•				26	
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	
SEL	Select	•	•				16	
SELEQZ	Select if Equal to Zero, else Zero	•	•				20	
SELEQZ	Select if Not Equal to Zero, else Zero	•	•				23	
SQRT	Square Root	•	•	**U**	**U**	**U**	4	
SUB	Subtract	•	•	•	**U**	**U**	1	

Table 7.21 Valid Formats for FPU Operations (Continued)

Mnemonic	Operation	Operand Fmt					COP1 Function Value	COP1X op4 Value
		Float			Fixed			
		S	D	PS[1]	W	L		
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.								

1. PS is deprecated by Release 6.

7.4 FPU Instruction Formats

An FPU instruction is a single 32-bit aligned word. FPU instruction formats are shown in Figures 7.1 through 7.10.

In these figures, variables are labeled in lowercase, such as *offset*. Constants are labeled in uppercase, as are numerals. Table 7.22 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.

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

Release 6 simplifies FPU instruction encoding, removing several formats as described in Table 7.22.

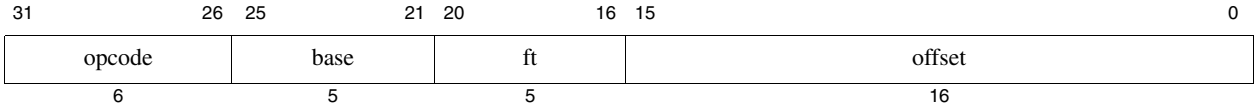
Table 7.22 FPU Instruction Fields

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. (Removed in Release 6)
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. (Removed in Release 6)
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.

Table 7.22 FPU Instruction Fields (Continued)

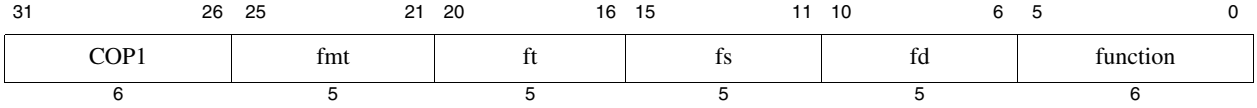
Field	Description
index	CPU register that holds the index address component for address calculations. (Removed in Release 6)
MOVC	Value in *function* field for a conditional move. There is one value for the instruction when *op*=COP1, another value for the instruction when *op*=SPECIAL. (Removed in Release 6)
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. (Removed in Release 6)

Figure 7.1 I-Type (Immediate) FPU Instruction Format



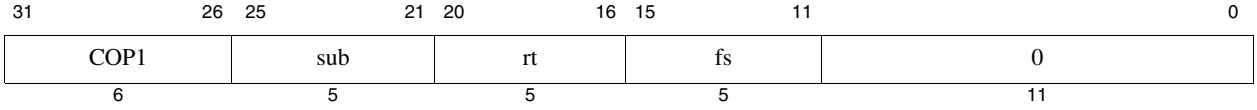
Immediate: Load/Store using register + offset addressing

Figure 7.2 R-Type (Register) FPU Instruction Format



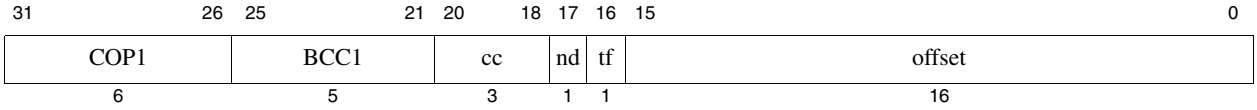
Register: Two-register and Three-register formatted arithmetic operations

Figure 7.3 Register-Immediate FPU Instruction Format



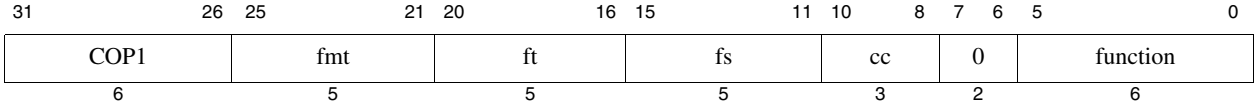
Register Immediate: Data transfer, CPU ↔ FPU register

Figure 7.4 Condition Code, Immediate FPU Instruction Format (Removed in Release 6)[1]



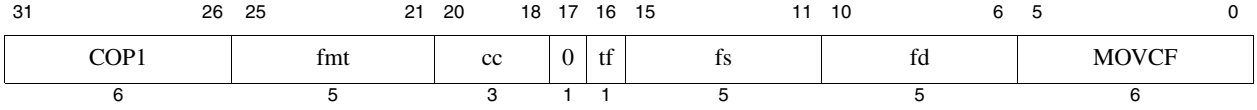
Condition Code, Immediate: Conditional branches on FPU cc using PC + offset

Figure 7.5 Formatted FPU Compare Instruction Format (Removed in Release 6)[2]



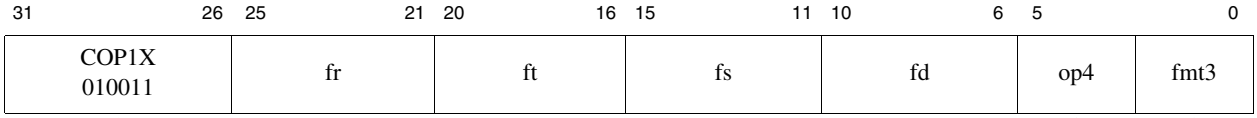
Register to Condition Code: Formatted FP compare

Figure 7.6 FP Register Move, Conditional Instruction Format (Removed in Release 6)[2]



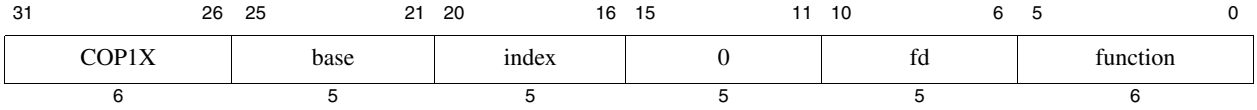
Condition Code, Register FP: FPU register move-conditional on FP, cc

Figure 7.7 Four-Register Formatted Arithmetic FPU Instruction Format (Removed in Release 6)[2]



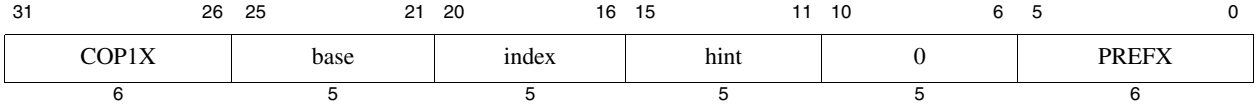
1. MIPS Release 6 uses the I-type (immediate) instruction format for its FPU conditional branches.
2. MIPS Release 6 removes uses FPU R-type instruction encodings for its FPU compare instruction (CMP.condn.fmt), its conditional move/select instructions (e.g., SEL.fmt), and its fused multiply add instructions (e.g., MADDF.fmt).

Figure 7.8 Register Index FPU Instruction Format (Removed in Release 6)[3]



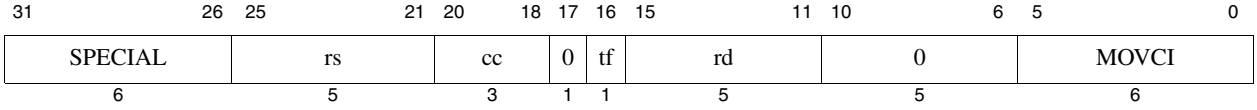
Register Index: Load and Store using register + register addressing

Figure 7.9 Register Index Hint FPU Instruction Format (Removed in Release 6)[4]



Register Index Hint: Prefetch using register + register addressing

Figure 7.10 Condition Code, Register Integer FPU Instruction Format (Removed in Release 6)[3]



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

3. MIPS Release 6 removes uses FPU R-type instruction encodings for its FPU compare instruction (CMP.condn.fmt), its conditional move/select instructions (e.g., SEL.fmt), and its fused multiply add instructions (e.g., MADDF.fmt).
4. MIPS Release 6 removes FPU Register Index addressing mode instructions.

Appendix A

Pipeline Architecture

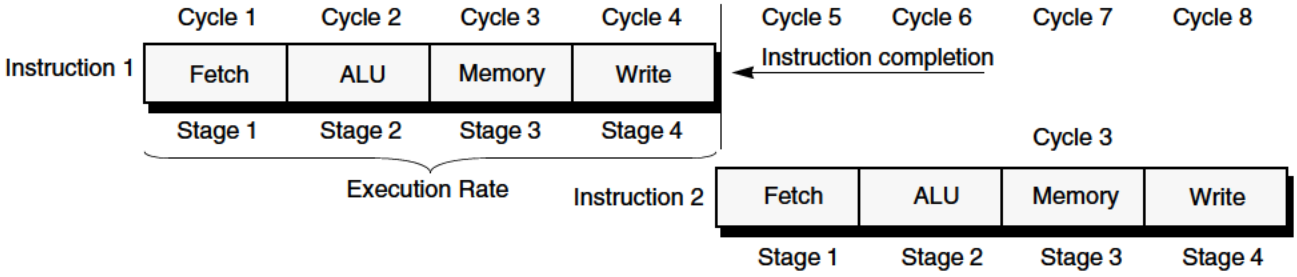
This Appendix describes the basic pipeline architecture, along with two types of improvements: superpipelines and superscalar pipelines. (Pipelining and multiple issuing are not defined by the ISA, but are implementation-dependent.)

A.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 A.1:

- Fetch
- Arithmetic operation
- Memory access
- Write back

Figure A.1 One-Deep Single-Completion Instruction Pipeline



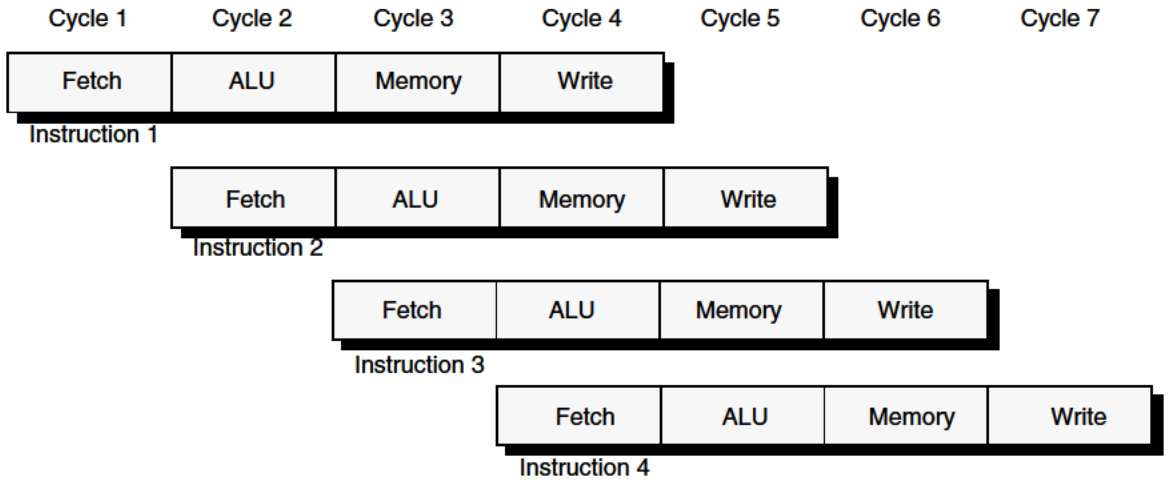
In the example shown in Figure A.1, each stage takes one processor clock cycle to complete. Thus it takes four clock cycles (ignoring delays or stalls) for the instruction to complete. In this example, the **execution rate** of the pipeline is one instruction every four clock cycles. Conversely, because only a single execution can be fetched before completion, only one stage is active at any time.

A.2 Parallel Pipeline

Figure A.2 illustrates a remedy for the **latency** (the time it takes to execute an instruction) inherent in the pipeline shown in Figure A.1.

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

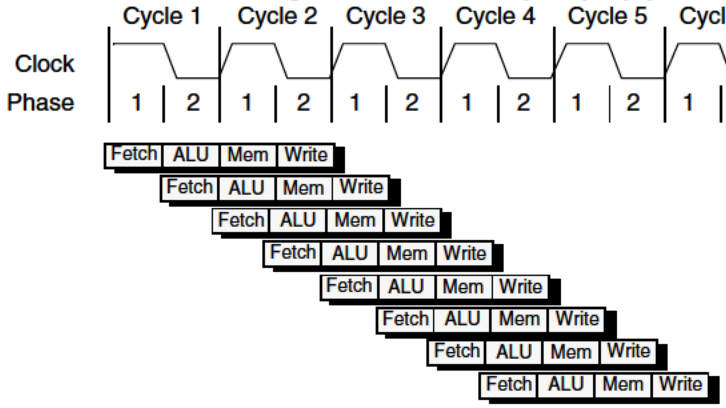
Figure A.2 Four-Deep Single-Completion Pipeline



A.3 Superpipeline

Figure A.3 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.

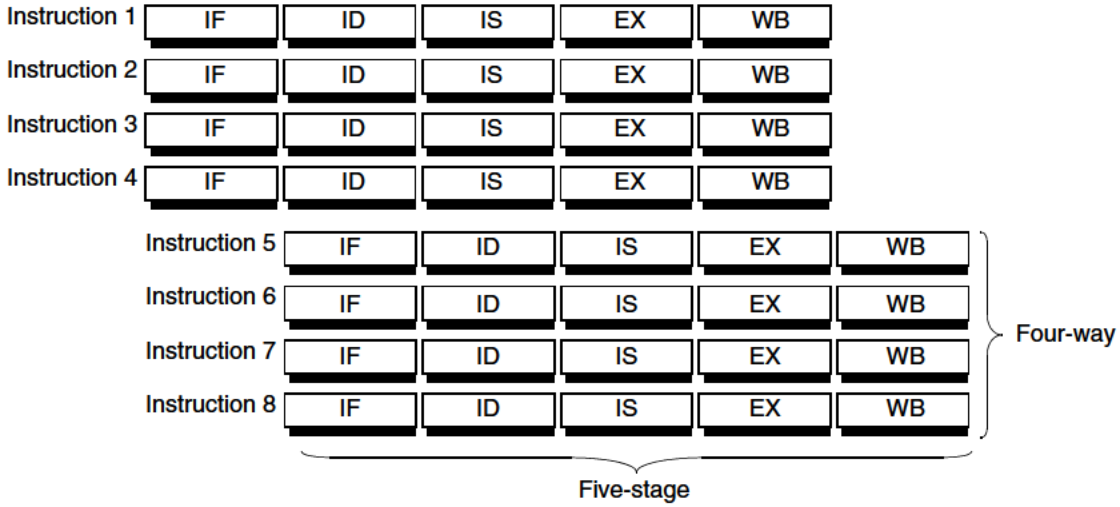
Figure A.3 Four-Deep Superpipeline



A.4 Superscalar Pipeline

A **superscalar** architecture also allows more than one instruction to be completed each clock cycle. Figure A.4 shows a four-way, five-stage superscalar pipeline.

Figure A.4 Four-Way Superscalar Pipeline



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

Appendix B

Misaligned Memory Accesses

Prior to Release 5 of the MIPS architectures, “natural” alignment was required of memory operands for most memory operations. Other than the Load Store Left (LSInstructions such as LWL and LWR. all memory operands were required to be were provided so that unaligned accesses can be performed by instruction sequences. To accomodaate umaligned memory access, the LWS and LSL instructions perform unaligned memory access using a sequence of instrucitons. In Release 5, the 128-bit vector memory accesses in load and store instructions (included in the MIPS SIMD Architecture) are not required to be naturally aligned.

Release 6 requires misaligned memory access support for all ordinary memory access instructions (for example, LW/SW, LWC1/SWC1). However, misaligned support is not provided for certain special memory accesses such as atomics (for example, LL/SC). Each instruction description in *Volume II* of the MIPS Architecture document set indicates whether misalignment support is provided.

Release 6 removes the unaligned memory access instructions (for example, LWL/LWR, LDL/LDR); these instructions are required to signal the Reserved Instruction exception. Release 6 also introduces the ALIGN instruction, which can be used following a pair of LW instructions that the programmer has ensured are aligned, in order to emulate a misaligned load without using LWL/LWR and without the possible performance penalty of a misaligned access.

The behavior, semantics, and architecture specifications of misaligned memory accesses are described in this appendix.

B.1 Terminology

This document uses the following terminology:

- “Unaligned” and “misaligned” generically refer to any memory value or reference not naturally aligned, but they may be used, for brevity, to refer to certain classes of memory access instructions as described below.
- The term “split” is used to refer to operations which cross important boundaries, whether architectural (for example, “page split” or “segment split”) or microarchitectural (for example, “cache line split”).
- Unaligned Load and Store Instructions
    - The MIPS Architecture specification has contained, since its beginning, special so-called Unaligned Load and Store instructions such as LWL/LWR and SWL/SWR (Load Word Left/Right, etc.)
    - When necessary, we will call these “explicit unaligned memory access instructions”, as distinct from “instructions that permit implicit misaligned memory accesses”, such as MSA vector loads and stores. However, where it is obvious from the context what we are talking about, we may say simply “unaligned” rather than the longer “explicit unaligned memory access instructions”, and “misaligned” rather than “instructions that permit implicit misaligned memory accesses”.
- Misaligned memory access instructions

    - More precisely, these are instructions that permit, and are required to support, implicit misaligned memory accesses.
    - Release 5 of the MIPS Architecture defines instructions, the MSA vector loads and stores, which may be aligned (for example, 128-bits on a 128 bit boundary), partially aligned (for example, “element aligned”, see below), or misaligned.
    - Release 6 makes almost all memory access instructions into “instructions that permit implicit misaligned memory accesses” (with exceptions such as LL/SC).
- Aligned memory access instructions
    - More precisely, these are instructions that do not permit, and which may be required to produce an exception, for misaligned addresses.
    - Prior to Release 6, all MIPS memory access instructions, for example, LW/SW, LL/SC, etc., required “natural alignment”, except for the “Unaligned Load and Store Instructions”, LWL/LWR and SWL/SWR.
    - Release 6 still requires alignment for a smaller set of memory access instructions, such as LL/SC.
- Misalignment is dynamic, known only when the address is computed (rather than static, explicit in the instruction as it is for LWL/LWR, etc.). We distinguish accesses for which the alignment is not yet known (“potentially misaligned”), from those whose alignment is known to be misaligned (“actually misaligned”), and from those for which the alignment is known to be naturally aligned (“actually aligned”). For example, LL/SC instructions are never potentially misaligned, i.e., are always actually aligned (if they do not trap). MSA vector loads and stores are potentially misaligned, although the programmer or compiler may prevent actual misalignment.

B.2 Hardware Versus Software Support for Misaligned Memory Accesses

Processors that implement versions of the MIPS Architectures prior to Release 5 require “natural” alignment of memory operands for most memory operations: apart from unaligned load and store instructions such as LWL/LWR and SWL/SWR, all memory accesses that are not naturally aligned are required to signal an Address Error exception.

Systems that implement Release 5 or higher of the MIPS Architectures require support for misaligned memory operands for MSA (MIPS SIMD Architecture) vector loads and stores (128-bit quantities). In Release 5, all misaligned memory accesses other than MSA continue to produce the Address Error exception.

Systems that implement Release 6 or higher of the MIPS Architectures (Release 6) require support for misaligned memory operands for almost all memory access instructions, including:

- Byte loads and stores of course cannot be misaligned: LB, LBE, LBU, LBUE. Nor can cache PREF instructions, etc.
- CPU loads and stores: LH/SH, LHU/SHU, LW/SW, LWU/SWU, and MIPS64 LD/SD
- The EVA versions of the above: LHE, LWE, MIPS64 LDE
- FPU loads and stores: LWC1/SWC1, LDC1/SDC1, LWXC1/SWXC1, LDXC1/SDXC1, LUXC1/SUXC1
- Coprocessor loads and stores: LWC2/SDC2, LDC2/SDC2

These are collectively called the “Ordinary Memory Accesses”.

In particular, misalignment support is NOT provided for the load-linked/store conditional instructions, namely, MIPS32 LL/SC, MIPS64 LLD/SCD, and EVA versions LLE/SCE. All such instructions continue to produce the Address Error exception if misaligned.

Note the phrasing “Systems that implement Release 5 or higher”. Processor hardware may provide varying degrees of support for misaligned accesses, producing the Address Error exception in certain cases. The software Address Error exception handler may then emulate the required misaligned memory access support in software. The term “systems that implement Release 5 or higher” includes such systems that combine hardware and software support. The processor in such a system by itself may not be fully Release 5 compliant because it does not support all misaligned memory references, but the combination of hardware and exception handler software may be.

Here are some examples of processor hardware providing varying degrees of support for misaligned accesses. The examples are named so that the different implementations can be discussed.

Full Misalignment Support: Some processors may implement all the required misaligned memory access support in hardware.

Trap (and Emulate) All Misaligneds: For example, it is permitted for a processor implementation to produce the Address Error exception for all misaligned accesses, with the appropriate exception handler software,

Trap (and Emulate) All Splits:
Intra-Cache-Line Misaligneds Support:
more accurately: **Misaligneds within aligned 64B regions Support:** For example, it is permitted for an implementation to perform misaligned accesses that fall entirely within a cache line in hardware, but to produce the Address Error exception for all cache line splits and page splits.

Trap (and Emulate Page) Splits:
Intra-Page Misaligneds Support:
more accurately: **Misaligneds within aligned 4KB regions Support:** For example, it is permitted for a processor implementation to perform cache line splits in hardware, but to produce the Address Error exception for page splits.

Distinct misaligned handling by memory type: For example, an implementation may perform misaligned accesses as described above for WB (Writeback) memory, but may produce the Address Error exception for all misaligned accesses involving the UC memory type.

Other mixes of hardware and software support are possible.

It is expected that Full Misaligned Support and Trap and Emulate Page Splits will be the most common implementations.

In general, actually misaligned memory accesses may be significantly slower than actually aligned memory accesses, even if an implementation provides Full Misaligned Support in hardware. Programmers and compilers should avoid actually misaligned memory accesses. Potentially but not actually misaligned memory accesses should suffer no performance penalty.

B.3 Detecting Misaligned Support

For Release 5 MSA misalignment support, it is sufficient to check that MSA is present, as defined by the appropriate reference manual[1]: i.e., support for misaligned MSA vector load and store instructions is required if the *Config3 MSAP* bit is set (CP0 Register 16, Select 3, bit 28).

For Release 6 misalignment support, it is sufficient to check the architecture revision level (CP0 $Config_{AR}$), as defined by the Privileged Resource Architecture.[2]

The need for software to emulate misaligned support as described in the previous section must be detected by an implementation specific manner, and is not defined by the Architecture.

B.4 Misaligned Semantics

B.4.1 Misaligned Fundamental Rules: Single-Thread Atomic, but not Multi-thread

The following principles are fundamental for the other architecture rules relating to misaligned support.

Architecture Rule B-1: Misaligned memory accesses are atomic with respect to a single thread (with limited exceptions noted in other rules).

(Indeed, this is true for all memory access instructions, and for all instructions in general.)

For example, all interrupts and exceptions are delivered either completely before or completely after a misaligned (split) memory access. Such an exception handler is not entered with part of a misaligned load destination register written and part unwritten. Similarly, it is not entered with part of a misaligned store memory destination written, and part unwritten.

As a counter example, uncorrectable ECC errors that occur halfway through a split store may violate single-thread atomicity.

Hardware page table walking is not considered to covered by single-thread atomicity.

Architecture Rule B-2: Memory accesses that are actually misaligned are not guaranteed to be atomic as observed from other threads, processors, and I/O devices.

B.4.2 Permissions and Misaligned Memory Accesses

Architecture Rule B-3: It must be permitted to access every byte specified by a memory access.

Architecture Rule B-4: It is NOT required that permissions, etc., be uniform across all bytes.

This applies to all memory accesses, but in particular applies to misaligned split accesses, which can cross page boundaries and/or other boundaries that have different permissions. It *IS* permitted for a misaligned, in particular a

1. For example, *MIPS® Architecture for Programmers, Volume IV-j: The MIPS32® SIMD Architecture Module*, Document Number MD00866; or the corresponding documents for other MIPS Architectures such as MIPS64®.
2. For example, *MIPS® Architecture Reference Manual, Volume III: The MIPS32® and microMIPS™ Privileged Resource Architecture*, Document Number MD00088; or the corresponding documents for other MIPS Architectures such as MIPS64®.

page split memory access, to cross permission boundaries, as long as the access is permitted by permissions on both sides of the boundary. Namely, it is not required that the permissions be identical for all parts, just that all parts are permitted.

Architecture Rule B-5: If any part of the misaligned memory access is not permitted, then the entire access must take the appropriate exception.

Architecture Rule B-6: If multiple exceptions arise for a given part of a misaligned memory access, then the same prioritization rules apply as for a non-misaligned memory access. For example, it is not acceptable for an exception to be reported with part of the access performed, e.g., part of a misaligned store written in memory, or part of a register written for a misaligned load.

Architecture Rule B-7: If different exceptions are mandated for different parts of a split misaligned access, it is UNPREDICTABLE which takes priority and is actually delivered. But at least one of them must be delivered.

Although it is permitted for misaligned accesses to be performed a byte at a time, in any order, this applies only to multiprocessor memory ordering issues, not to exception reporting.

Architecture Rule B-8: When an exception is delivered for a split misaligned address, EPC points to either the instruction that performed the split misaligned access, or to the branch in whose delay slot or forbidden slot the former instruction lies, in the usual manner.

Architecture Rule B-9: The address reported by BadVaddr on address error, for example, for a misaligned access that is not directly supported by hardware, but which will be emulated by a trap handler, must be the lowest byte virtual address associated with the misaligned access.

Architecture Rule B-10: The address reported by BadVaddr on page permission or TLB miss exceptions must be a byte address in the misaligned access for a page on which the exception is reported, but may be any such byte address. It is not required to be the lowest.

Architecture Rule B-11: If both parts of a page split misaligned access produce the same exception, it is UNPREDICTABLE which takes priority. BadVaddr is either the smallest byte virtual address in the first part, i.e., the start of the misaligned access, or the smallest byte address in the second part, i.e., the start of the second page.

This permits page fault handlers to be oblivious to misalignment: they just remedy the page fault reported by BadVaddr, and return. There is no architectural mechanism to guarantee forward progress; the system implementation, both hardware and software, must arrange for forward progress. For example, on a single-threaded CPU, the TLB associativity must be such that all TLB entries relevant to page splits can be resident. If this is impossible, for example, on a multithreaded CPU without partitioned TLBs, it may be necessary for the exception handlers to emulate the instruction for which inability to make forward progress is detected.

For example, if an access is split across two pages, the first part of the split is permitted, but the second part is not permitted, then the BadVaddr reported must be for the smallest byte address in the second part. For example, if a misaligned load is a page split, and one part of the load is to a page marked read-only, while the other is to a page marked invalid, the entire access must take the TLB Invalid exception. The destination register will NOT be partially written. BadVaddr will contain the lowest byte address

For example, if a misaligned store is a page split, and one part of the store is to a page marked writable, while the other part is to a page marked read-only, the entire store must take the TLB Modified exception. It is NOT permitted to write part of the access to memory, but not the other part.

For example, if a misaligned memory access is a page split, and part is in the TLB and the other part is not - if software TLB miss handling is enabled then none of the access shall be performed before the TLB Refill exception is entered.

For example, if a misaligned load is a page split, and one part of the load is to a page marked read-only, while the other is to a page marked read-write, the entire access is permitted. Namely, a hardware implementation MUST perform the entire access. A hardware/software implementation may perform the access or take an Address Error exception, but if it takes an Address Error exception trap no part of the access may have been performed on arrival to the trap handler.

B.4.3 Misaligned Memory Accesses Past the End of Memory

Architecture Rule B-12: Misaligned memory accesses past the end of virtual memory are permitted, and behave as if a first partial access was done from the starting address to the virtual address limit, and a second partial access was done from the low virtual address for the remaining bytes.

For example, an N byte misaligned memory access (N=16 for 128-bit MSA) starting M bytes below the end of the virtual address space “VMax” will access M bytes in the range [VMax-M+1,VMax], and in addition will access N-M bytes starting at the lowest virtual address “VMin”, the range [VMin, VMin+N-M-1].

For example, for 32-bit virtual addresses, VMin=0 and VMax = 2^{32}-1, and an N byte access beginning M bytes below the top of the virtual address space expands to two separate accesses as follows:
$2^{32} - \mathrm{M} \Rightarrow [2^{32}\text{-M},2^{32} - 1] \cup [0, 0 + N - M]$

For example, for 64 bit virtual addresses, VMin=0 and VMax = 2^{64}-1, and an N byte access beginning M bytes below the top of the virtual address space expands to two separate accesses as follows:
$2^{64} - \mathrm{M} \Rightarrow [2^{64}\text{-M},2^{64} - 1] \cup [0, 0 + N - M]$

Similarly, both 32 and 64 bit accesses can cross the corresponding signed boundaries, for example, from, 0x7FFF_FFFF to 0x8000_0000 or from 0x7FFF_FFFF_FFFF_FFFF to 0x8000_0000_0000_0000.

Architecture Rule B-13: Beyond the wrapping at 32 or 64 bits mentioned, above, there is no special handling of accesses that cross MIPS segment boundaries, or which exceed SEGBITS within a MIPS segment.

For example, a 16 byte MSA access may begin in `xuseg` with a single byte at address 0x3FFF_FFFF_FFFF_FFFF and cross to `xsseg`, for example, 15 bytes starting from 0x4000_000_0000_0000 - assuming consistent permissions and CCAs.

Architecture Rule B-14: Misaligned memory accesses must signal Address Error exception if any part of the access would lie outside the physical address space.

For example, if in an unmapped segment such as kseg0, and the start of the misaligned is below the PABITS limit, but the access size crosses the PABITS limit.

B.4.4 TLBs and Misaligned Memory Accesses

A specific case of rules stated above:

Architecture Rule B-15: if any part of a misaligned memory access involves a TLB miss, then none of the access shall be performed before the TLB miss handling exception is entered.

Here “performed” means the actual store that changes memory or cache data values, or the actual load that writes a destination register, or side effects of loads related to memory mapped I/O. It does not refer to microarchitectural side effects such as changing the state of a cache line from M in another processor to S locally or to TLB state.

Note: this rules does NOT disallow emulating misaligned memory accesses via a trap handler that performs the access a byte at a time, even though a TLB miss may occur for a later byte after an earlier byte has been written. Such a trap handler is emulating the entire misaligned. A TLB miss in the emulation code will return to the emulation code, not to the original misaligned memory instruction.

However, this rule DOES disallow handling permissions errors in this manner. Write permission must be checked in advance for all parts of a page split store.

Architecture Rule B-16: Misaligned memory accesses are not atomic with respect to hardware page table walking for TLB miss handling (as is added in MIPS Release 5).

Overall, TLBs, in particular hardware page table walking, are not considered to be part of “single-thread atomicity”, and hardware page table walks are not ordered with the memory accesses of the loads and stores that trigger them.

For example, the different parts of a split may occur at different times, and speculatively. If another processor is modifying the page tables without performing a TLB shutdown, the TLB entries found for a split may not have both been present in the TLBs at the same time. On a system with hardware page table walking, the page table entries for a split may not have both been present in the page tables in memory at the same time.

For example, on an exception triggered by a misaligned access, it is UNPREDICTABLE which TLB entries for a page split are in the TLB: both, one but not the other, or none.

Implementations must provide mechanisms to accommodate all parts of a misaligned load or store in order to guarantee forward progress. For example, a certain minimum number of TLB entries may be required for the split parts of a misaligned memory access, and/or associated software TLB miss handlers or hardware TLB miss page table walkers. Other such mechanisms may not require extra TLB entries.

Architecture Rule B-17: Misaligned memory accesses are not atomic with respect to setting of PTE access and dirty bits.

For example, if a hardware page table walker sets PTE dirty bit for both parts of a page split misaligned store, then it may be possible to observe one bit being set while the other is still not set.

Architecture Rule B-18: Misaligned memory accesses that affect any part of the page tables in memory that are used in performing the virtual to physical address translation of any part of the split access are UNPREDICTABLE.

For example, a split store that writes one of its own PTEs - whether the hardware page table walker PTE, or whatever data structure a software PTE miss handler uses. (This means that a simple Address Error exception handler can implement misaligneds without having to check page table addresses.)

B.4.5 Memory Types and Misaligned Memory Accesses

Architecture Rule B-19: Misaligned memory accesses are defined and are expected to be used for the following CCAs: WB (Writeback) and UCA (Uncached Accelerated), i.e., write combining.

Architecture Rule B-20: Misaligned memory accesses are defined for UC. Instructions that are potentially misaligned, but which are not actually misaligned, may safely be used with UC memory including MMIO. But instructions which are actually misaligned should not be used with MMIO - their results may be UNPREDICTABLE or worse.

Misaligned memory accesses are defined for the UC (Uncached) memory type, but their use is recommended only for ordinary uncached memory, DRAM or SRAM. The use of misaligned memory accesses is discouraged for uncached memory mapped I/O (MMIO) where accesses have side effects, because the specification of misaligned memory accesses does not specify the order or the atomicity in which the parts of the misaligned access are performed, which it makes it very difficult to use these accesses to control memory-mapped I/O devices with side effects.

Architecture Rule B-21: Misaligned memory accesses that cross two different CCA memory types are UNPREDICTABLE. (Reasons for this may include crossing of page boundaries, segment boundaries, etc.)

Architecture Rule B-22: Misaligned memory accesses that cross page boundaries, but with the same memory type in both pages, are permitted.

Architecture Rule B-23: Misaligned memory accesses that cross segment boundaries are well defined, so long as the memory types in both segments are the same and are otherwise permitted.

B.4.6 Misaligneds, Memory Ordering, and Coherence

This section discusses single and multithread atomicity and multithread memory ordering for misaligned memory accesses. However, it does not address issues for potentially misaligned memory references. Documents such as the *MIPS Coherence Protocol Specification* define this behavior.[3]

B.4.6.1 Misaligneds are Single-Thread Atomic

Recall the first fundamental rule of misaligned support concerning single-thread atomicity:

Architecture Rule B-1: “Misaligned memory accesses are atomic with respect to a single thread (with limited exceptions noted in other rules).” on page 129.

For example, all interrupts and exceptions are delivered either completely before or completely after a misaligned (split) memory access. Such an exception handler is not entered with part of a misaligned load destination register written, and part unwritten. Similarly, it is not entered with part of a misaligned store memory destination written, and part unwritten.

Architecture Rule B-24: However, an implementation may not be able to enforce single-thread atomicity for certain error conditions.

Architecture Rule B-25: For example, single-thread atomicity for a misaligned, cache line or page split store, MAY be violated when an uncorrectable ECC error detected when performing a later part of a misaligned, when an part has already been performed, updating memory or cache.

Architecture Rule B-26: Nevertheless, implementations should avoid violating single-thread atomicity whenever possible, even for error conditions.

Here are some exceptional or error conditions for which violating single-thread atomicity for misaligneds is NOT acceptable: any event involving instruction access rather than data access, Debug data breakpoints, Watch address match, Address Error, TLB Refill, TLB Invalid, TLB Modified, Cache Error on load or LL, Bus Error on load or LL.

Machine Check exceptions (a) are implementation-dependent, (b) could potentially include a wide number of processor internal inconsistencies. However, at the time of writing the only Machine Check Exceptions that are defined

3. For example, *MIPS Coherence Protocol Specification* (AFP Version), Document Number MD00605. Updates and revisions of this document are pending.

are (a) detection of multiple matching entries in the TLB, and (b) inconsistencies in memory data structures encountered by the hardware page walker page table. Neither of these should cause a violation of single-thread atomicity for misaligneds. In general, no errors related to virtual memory addresses should cause violations of single-thread atomicity.

Architecture Rule B-27: Reset (Cold Reset) and Soft Reset are not required to respect single-thread atomicity for misaligned memory accesses. For example, Reset may be delivered when a store is only partly performed.

However, implementations are encouraged to make Reset and, in particular, Soft Reset, single instruction atomic whenever possible. For example, a Soft Reset may be delivered to a processor that is not hung, when a misaligned store is only partially performed. If possible, the rest of the misaligned store should be performed. However, if the processor is stays hung with the misaligned store only partially performed, then the hang should time out and reset handling be completed.

Non-Maskable Interrupt (NMI) is required to respect single-thread atomicity for misaligned memory accesses, since NMIs are defined to only be delivered at instruction boundaries.

B.4.6.2 Misaligneds are not Multiprocessor/Multithread Atomic

Recall the second fundamental rule of misaligneds - lack of multiprocessor atomicity:

Architecture Rule B-2: “Memory accesses that are actually misaligned are not guaranteed to be atomic as observed from other threads, processors, and I/O devices.” on page 129.

The rules in this section provide further detail.

Architecture Rule B-28: Instructions that are potentially misaligned memory accesses but which are not actually misaligned may be atomic, as observed from other threads, processors, or I/O devices.

The overall Misaligned Memory Accesses specification does not address issues for potentially but not actually misaligned memory references. Documents such as the *MIPS Coherence Protocol Specification* define such behavior

Architecture Rule B-29: Actually misaligned memory accesses may be performed in more than one part. The order of these parts is not defined.

Architecture Rule B-30: It is UNPREDICTABLE and implementation-dependent how many parts may be used to implement an actually misaligned memory access.

For example, a page split store may be performed as two separate accesses, one for the low part, and one for the high part.

For example, a misaligned access that is not split may be performed as a single access.

For example, or a misaligned access - any misaligned access, not necessarily a split - may be performed a byte at a time.

Although most of this section has been emphasizing behavior that software cannot rely on, we can make the following guarantees:

Architecture Rule B-31: every byte written in a misaligned store will be written once and only once.

Architecture Rule B-32: a misaligned store will not be observed to write any bytes that are not specified: in particular, it will not do a read of memory that includes part of a split, merge, and then write the old and new data back.

Note the term “observed” in the rule above. For example, memory and cache systems using word or line oriented ECC may perform read-modify-write in order to write a subword such as a byte. However, such ECC RMWs are atomic from the point of view of other processors, and do not affect bytes not written.

See section B.6 “Misalignment and MSA Vector Memory Accesses” on page 139, for a discussion of element atomicity as applies to misaligned MSA vector memory accesses.

B.4.6.3 Misaligneds and Multiprocessor Memory Ordering

Preceding sections have defined misaligned memory accesses as having single-thread atomicity but not multithread atomicity. Furthermore, there are issues related to memory ordering overall:

Architecture Rule B-33: Instructions that are potentially but not actually misaligned memory accesses comply with the MIPS Architecture rules for memory consistency, memory ordering, and synchronization.

This section Misaligned Memory Accesses does not address issues for potentially but not actually misaligned memory references. Documents such as the *MIPS Coherence Protocol Specification* define such behavior.

Architecture Rule B-34: The split parts of actually misaligned memory accesses obey the memory ordering rules of the MIPS architecture.

Although actually misaligned memory references may be split into several smaller references, as described in previous sections, these smaller references behave as described for any memory references in documents such as the *MIPS Coherence Protocol Specification*. In particular, misaligned subcomponent references respect the ordering and completion types of the SYNC instruction, legal and illegal sequences described in that document.

B.5 Pseudocode

Pseudocode can be convenient for describing the operation of instructions. Pseudocode is not necessarily a full specification, since it may not express all error conditions, all parallelism, or all non-determinism - all behavior left up to the implementation. Also, pseudocode may over-specify an operation, and appear to make guarantees that software should not rely on.

The first stage pseudocode provides functions `LoadPossiblyMisaligned` and `StorePossiblyMisaligned` that interface with other pseudocode via virtual address `vAddr`, the memory request size `nbytes` (=16 for 128b MSA), and arrays of byte data `inbytes[nbytes]` and `inbytes[nbytes]`.

The byte data is assumed to be permuted as required by the Big and Little endian byte ordering modes as required by the different instructions - thus permitting the pseudocode for misalignment support to be separated from the endianness considerations. Namely, `outbytes[0]` contains the value that a misaligned store will write to address vAddr+0, and so on.

The simplest thing that could possibly work would be to operate as follows:

```
for i in 0 .. nbytes-1
    (pAddr, CCA) ← AddressTranslation (vAddr+i, DATA, LOAD)
    inbytes[i] ← LoadRawMemory (CCA, nbytes, pAddr, vAddr+i, DATA)
endfor

for i in 0 .. nbytes-1
    (pAddr, CCA) ← AddressTranslation (vAddr+i, DATA, STORE)
    StoreRawMemory (CCA, 1, outbytes[i], pAddr, vAddr+i, DATA)
endfor
```

but this simplest possible pseudocode does not express the atomicity constraints and certain checks.

B.5.1 Pseudocode Distinguishing Actually Aligned from Actually Misaligned

The top-level pseudocode functions `LoadPossiblyMisaligned/StorePossiblyMisaligned` take different paths depending on whether actually aligned or actually misaligned. This reflects the fact that aligned and misaligned have different semantics, different atomicity properties, etc.

Figure B.1 LoadPossiblyMisaligned/StorePossiblyMisaligned Pseudocode

```
inbytes[nbytes] ← LoadPossiblyMisaligned(vaddr, nbytes)
   if naturally_aligned(vaddr,nbytes)
      return LoadAligned(vaddr,nbytes)
   else
      return LoadMisaligned(caddr,nbytes)
endfunction LoadPossiblyMisaligned

StorePossiblyMisaligned(vaddr, outbytes[nbytes])
   if naturally_aligned(vaddr,nbytes)
      StoreAligned(vaddr,nbytes)
   else
      StoreMisaligned(caddr,nbytes)
endfunction StorePossiblyMisaligned
```

B.5.2 Actually Aligned

The aligned cases are very simple, and are defined to be a single standard operation from the existing pseudocode repertoire (except for byte swapping), reflecting the fact that actually aligned memory operations may have certain atomicity properties in both single and multithread situations.

Figure B.2 LoadAligned / StoreAligned Pseudocode

```
inbytes[nbytes] ← LoadAligned(vaddr, nbytes)
   assert naturally_aligned(vaddr,nbytes)
   (pAddr, CCA) ← AddressTranslation (vAddr, DATA, LOAD)
   return inbytes[] ← LoadRawMemory (CCA, nbytes, pAddr, vAddr, DATA)
endfunction LoadAligned

StoreAligned(vaddr, outbytes[nbytes])
   assert naturally_aligned(vaddr,nbytes)
   (pAddr, CCA) ← AddressTranslation (vAddr, DATA, LOAD)
   StoreRawMemory (CCA, nbytes, outbytes, pAddr, vAddr, DATA)
endfunction StoreAligned
```

B.5.3 Byte Swapping

The existing pseudocode uses functions LoadMemory and StoreMemory to access memory, which are declared but not defined. These functions implicitly perform any byte swapping needed by the Big and Little endian modes of the MIPS processor, which is acceptable for naturally aligned scalar data memory load and store operations. However, with vector operations and misaligned support, it is necessary to assemble the bytes from a memory load instruction, and only then to byte swap them—i.e., byte swapping must be exposed in the pseudocode, and conversely for stores.

Figure B.3 LoadRawMemory Pseudocode Function

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

   /* like the original pseudocode LoadMemory, except no byteswapping */

   /* MemElem:  A vector of AccessLength bytes, in memory order. */
   /* CCA:      Cacheability&CoherencyAttribute=method used to access caches */
   /*           and memory and resolve the reference */

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

endfunction LoadRawMemory
```

Figure B.4 StoreRawMemory Pseudocode Function

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

   /* like the original pseudocode StoreMemory, except no byteswapping */

   /* CCA:      Cacheability&Coherency Attribute, the method used to access */
   /*           caches and memory and resolve the reference. */
   /* AccessLength: Length, in bytes, of access */
   /* MemElem:  A vector of AccessLength bytes, in memory order. */
   /* pAddr:    physical address */
   /* vAddr:    virtual address */

endfunction StoreRawMemory
```

Helper functions for byte swapping according to endianness:

Figure B.5 Byte Swapping Pseudocode Functions

```
outbytes[nbytes] ← ByteReverse(inbytes[nbytes], nbytes)
   for i in 0 .. nbytes-1
      outbytes[nbytes-i] ← inbytes[i]
   endfor
   return outbytes[]
endfunction ByteReverse

outbytes[nbytes] ← ByteSwapIfNeeded(inbytes[nbytes], nbytes)
   if BigEndianCPU then
      return ByteReverse(inbytes)
   else
      return inbytes
endfunction ByteSwapIfNeeded
```

B.5.4 Pseudocode Expressing Most General Misaligned Semantics

The misaligned cases have fewer constraints and more implementation freedom. The very general pseudocode below makes explicit some of the architectural rules that software can rely on, as well as many things that software should NOT rely on: lack of atomicity both between and within splits, etc. However, we emphasize that only the behavior guaranteed by the architecture rules should be relied on.

Figure B.6 LoadMisaligned most general pseudocode

```
inbytes[nbytes] ← LoadMisaligned(vaddr, nbytes)
   if any part of [vaddr,vaddr+nbytes) lies outside valid virtual address range
      then SignalException(...)
   for i in 0 .. nbytes-1
       (pAddr[i], CCA[i]) ← AddressTranslation (vAddr+i, DATA, LOAD)
   if any pAddr[i] is invalid or not permitted then SignalException(...)
   if any CCA[i] != CCA[j], where i, j are in [0,nbytes) then UNPREDICTABLE
   loop // in any order, and possibly in parallel
      pick an arbitrary subset S of [0,nbytes) that has not yet been loaded
      load inbytes[S] from memory with the corresponding CCA[i], pAddr[i], vAddr+i
      remove S from consideration
   until set of byte numbers remaining unloaded is empty.
   return inbytes[]
endfunction LoadMisaligned
// ...similarly for StoreMisaligned...
```

B.5.5 Example Pseudocode for Possible Implementations

This section provides alternative implementations of `LoadMisaligned` and `StoreMisaligned` that emphasize some of the permitted behaviors.

It is emphasized that these are not specifications, just examples. Examples to emphasize that particular implementations of misaligneds may be permitted. But these examples should not be relied on. Only the guarantees of the architecture rules should be relied on. The most general pseudocode seeks to express these in the most general possible form.

B.5.5.1 Example Byte-by-byte Pseudocode

The simplest implementation is to operate byte-by-byte. Here presented more formally than above, because the separate byte loads and stores expresses the desired lack of guaranteed atomicity (whereas for `{Load,Store}PossiblyMisaligned` the separate byte loads and stores would not express possible guarantees of atomicity). Similarly, the pseudocode translates the addresses twice, a first pass to check if there are any permissions errors, a second pass to actually use ordinary stores. UNPREDICTABLE behavior if the translations change between the two passes.

This pseudocode tries to indicate that it is permissible to use such a two-phase approach in an exception handler to emulate misaligneds in software. It is not acceptable to use a single pass of byte-by-byte stores, unless split stores that are half completed can be withdrawn, transactionally. However, it is not required to save the translations of the first pass to reuse in the second pass (which would be extremely slow); if virtual address translations or permissions change between the first and second pass, it is acceptable to produce a partially written memory result.

Figure B.7 Byte-by-byte Pseudocode for LoadMisaligned / StoreMisaligned

```
inbytes[nbytes] ← LoadMisaligned(vaddr, nbytes)
   for i in 0 .. nbytes-1
       (ph1.pAddr[i], ph1.CCA[i]) ← AddressTranslation (vAddr+i, DATA, LOAD)
   /* ... checks ... */
   for i in 0 .. nbytes-1
      (ph2.pAddr[i], ph2.CCA[i]) ← AddressTranslation (vAddr+i, DATA, LOAD)
      if ph1.pAddr[i] != ph2.addr or ph1.CCA[i] != ph2.CCA[i] then UNPREDICTABLE
      inbytes[i] ← LoadRawMemory(ph2.CCA[i], nbytes, ph2.pAddr[i], vAddr+i, DATA)
   return inbytes[]
endfunction LoadMisaligned
```

```
StoreMisaligned(vaddr, outbytes[nbytes])
   for i in 0 .. nbytes-1
        (ph1.pAddr[i], ph1.CCA[i]) ← AddressTranslation (vAddr+i, DATA, LOAD)
   /* ... checks ... */
   for i in 0 .. nbytes-1
      (ph2.pAddr[i], ph2[i].CCA) ← AddressTranslation (vAddr+i, DATA, LOAD)
      if ph1.pAddr[i] != ph2.addr or ph1.CCA[i] != ph2.CCA[i] then UNPREDICTABLE
      StoreRawMemory(ph2[i].CCA, nbytes, outbytes[i], ph2.pAddr[i], vAddr+i, DATA)
endfunction StoreMisaligned
```

B.5.5.2 Example Pseudocode Handling Splits and non-Splits Separately

A more aggressive implementation, which is probably the preferred implementation on typical hardware:

- If a misaligned request is not split, it is performed as a single operation
- If a misaligned request is split, it is performed as two separate operations, with cache line and page splits handled separately.

B.6 Misalignment and MSA Vector Memory Accesses

B.6.1 Semantics

Misalignment support is defined by Release 5 of the MIPS Architecture for MSA (MIPS SIMD Architecture)[4] vector load and store instructions, including Vector Load (LD.df), Vector Load Indexed (LDX.df), Vector Store (ST.df) and Vector Store Indexed (STX.df). Each vector load and store has associated with it a data format, ".df", which can be byte/halfword/word/doubleword (B/H/W/D) (8/16/32/64 bits). The data format defines the vector element size.

The data format is used to determine Big-endian versus Little-endian byte swapping, and also influences multiprocessor atomicity as described here.

Architecture Rule B-35: Vector memory reference instructions are single-thread atomic, as defined above.

Architecture Rule B-36: Vector memory reference instructions have element atomicity.

If the vector is aligned on the element boundary, i.e., if the vector address is =0 modulo 2, 4, 8 for H/W/D respectively, then for the purposes of multiprocessor memory ordering the vector memory reference instruction can be considered a set of vector element memory operations. The vector element memory operations may be performed in any order, but each vector element operation, since naturally aligned, has the atomicity of the corresponding scalar.

In MIPS32 Release 5, all 16- and 32-bit scalar accesses are defined to be atomic; for example, each of the 32-bit elements of a word vector loaded using LD.W would be atomic. However, in MIPS32 Release 5, 64-bit accesses are not defined to be atomic, so LD.D would not have element atomicity.

In MIPS64 Release 6, all 16-, 32-, and 64-bit scalar accesses are atomic. So vector LD.H, LD.W, LD.D, and the corresponding stores would be element atomic.

4. *MIPS® Architecture Reference Manual, Volume IV-j: The MIPS32® SIMD Architecture Module*, MIPS Document Number MD00867.

All of the rules in sections B.4.2 "Permissions and Misaligned Memory Accesses", B.4.4 "TLBs and Misaligned Memory Accesses", B.4.5 "Memory Types and Misaligned Memory Accesses", and B.4.6.1 "Misligneds are Single-Thread Atomic" apply to the vector load or store instructions as a whole.

For example, a misaligned vector load instruction will never leave its vector destination register half written, if part of a page split succeeds and the other part takes an exception. It is either all done, or not at all.

For example, misaligned vector memory references that partly fall outside the virtual address space are UNPREDICTABLE.

However, the multiprocessor and multithread oriented rules of section B.4.6.2 "Misligneds are not Multiprocessor/Multithread Atomic" and B.4.6.3 "Misligneds and Multiprocessor Memory Ordering" do NOT apply to the vector memory reference instruction as a whole. These rules only apply to vector element accesses.

In fact, all of the rules of B.4 "Misaligned Semantics" apply to all vector element accesses - except where "overridden" for the vector as a whole.

For example, a misaligned vector memory reference that crosses a memory type boundary, for example, which is page split between WB and UCA CCAs, is UNPREDICTABLE. Even though, if the vector as whole is vector element aligned, no vector element crosses such a boundary, so that if the vector element memory accesses were considered individually, each would be predictable.

These instructions specify an element type, for example, ST.B, ST.H, ST.W, ST.D for Vector Store Byte/Halfword/Word/Doubleword respectively. If the memory address is naturally aligned for the element type, then atomicity is guaranteed for each element: for example, any element stored is entirely seen or entirely not seen, but one will never see half an such an element written, and half unwritten. One may, however, see some elements of the vector written, and some not written, even if the overall vector is naturally aligned. If the misaligned address is not naturally aligned for the element type, then the atomicity rules for ordinary memory accesses apply to the vector elements.

Note that Architecture Rule B-36 implies that smaller element accesses such as ST.B should not be used for larger accesses such as ST.D. Endianness considerations also imply that this is inadvisable.

B.6.2 Pseudocode for MSA Memory Operations with Misalignment

The MSA specification uses the pseudocode functions shown in Figure B.8 to access memory.

Figure B.8 LoadTYPEVector / StoreTYPEVector used by MSA specification

```
function LoadTYPEVector(ts, a, n)
   /* Implementation defined
      load ts, a vector of n TYPE elements
      from virtual address a.
   */
endfunction LoadTYPEVector

function StoreTYPEVector(tt, a, n)
   /* Implementation defined
      store tt, a vector of n TYPE elements
      to virtual address a.
   */
endfunction StoreTYPEVector
```

where `TYPE` = Byte, Halfword, Word, Doubleword
For example, LoadByteVector, LoadHalfwordVector, etc.

These can be expressed in terms of the misaligned pseudocode operations as shown in Figure B.9 and Figure B.10 by passing the TYPE (Byte, Halfword, Word, DoubleWord) as a parameter:

Figure B.9 Pseudocode for LoadVector

```
function LoadVector(vregdest, vAddr, nelem, TYPE)
   vector_wide_assertions(vAddr, nelem, TYPE)
   for all i in 0 to nelem-1 do /* in any order, any combination */
      rawtmp[i] ← LoadPossiblyMisaligned( vAddr + i*sizeof(TYPE), sizeof(TYPE) )
      bstmp[i] ← ByteSwapIfNeeded( rawtmp[i], sizeof(TYPE) )
      /* vregdest.TYPE[i] ← bstmp[i] */
      vregdest_nbits(TYPE)*i+nbits(TYPE)-1..nbits(TYPE)*i = bstmp[i]
   endfor
endfunction LoadVector
```

Figure B.10 Pseudocode for StoreVector

```
function StoreVector(vregsrc, vAddr, nelem, TYPE)
   vector_wide_assertions(vAddr, nelem, TYPE)
   for i in 0 .. nelem-1 /* in any order, any combination */
      bstmp[i] ← vregsrc_nbits(TYPE)*i+nbits(TYPE)-1..nbits(TYPE)*i
      rawtmp[i] ← ByteSwapIfNeeded( rawtmp[i], sizeof(TYPE) )
      StorePossiblyMisaligned( vAddr + i*sizeof(TYPE), sizeof(TYPE) )
   endfor
endfunction StoreVector
```

Appendix C

Revision History

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 2.4, "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 & endianness 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.
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, 2011	• There is no persistent 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.

Revision	Date	Description
3.50	September 20,2012	• Mention EVA load, store instructions • Define Architecture version of UCA. • IEEE2008, MAC2008, ABS2008, NAN2008 status bits for FPU. • Mention SegCtl, TLBInv*, EVA in Intro.
5.00	December 14, 2012	• R5 changes - mention MSA and VZ modules • R5 change - DSP and MT are now modules • Generated QNAN values - changed to use more common bit patterns
5.01	December 15, 2012	• No technical content change: • Updated cover for logos • Updated copyright text.
5.02	April 12, 2013	• R5 changes: FR=1 64-bit FPU register model required is required, if floating point is supported. Section 2.1.2.4 MIPSr5 Architecture. Section 2.2 Compliance and Subsetting. Section 2.8.5 FPU Registers. Chapter 5 Overview of the FPU Instruction Set: Section 5.1 Binary Compatibility. Section 5.5 Floating Point register Types. Section 5.5.1 FPU Register Models. • R5 change: if any R5 feature, other features must be R5. E.g. if VZ or MSA is implemented, then if floating point is implemented then FR=1 must be implemented. Section 2.2 Compliance and Subsetting. • R5 change retroactive to R3: removed $FCSR_{MCA2008}$ bit: no architectural support for fused multiply add with no intermediate rounding. Section 2.1.2.3 MIPSr3 Architecture. Table 5.4 FIR Register Field Descriptions, HAS2008 bit. Figure 5-12 FCSR register Format: MAC2008 bit removed. Section 5.9.2 Arithmetic Instructions: paragraph titled “Arithmetic and rounding behavior”. • R5 change: UFR (User mode FR changing): UFR, UNFR, FIR.UFRP, CTC1 and CFC1 changes. Section 5.6 Floating Point Control Registers (FCRs) - UFR and UNFR FCR numbers; Figure 5-11 FIR Register Format, Table 5.6 FIR Register Field Descriptions - UFRP bit; Section 5.6.2 UFR Register and Section 5.6.3 UNFR Register.

Revision	Date	Description
5.03	August 21, 2013	• Resolved inconsistencies with regards to the availability of instructions in MIPS32r2: MADD.fmt family (MADD.S, MADD.D, NMADD.S, NMADD.D, MSUB.S, MSUB.D, NMSUB,S, NMSUB.D), RECIP.fmt family (RECIP.S, RECIP.D, RSQRT.S, RSQRT.D), and indexed FP loads and stores (LWXC1, LDXC1, SWXC1, SDXC1). The appendix section A.2 “Instruction Bit Encoding Tables”, shared between Volume I and Volume II of the ARM, was updated, in particular the new upright delta Δ mark is added to Table A.2 “Symbols Used in the Instruction Encoding Tables”, replacing the inverse delta marking ∇ for these instructions. Similar updates made to microMIPS’s corresponding sections. Instruction set descriptions and pseudocode in Volume II, Basic Instruction Set Architecture, updated. These instructions are required in MIPS32r2 if an FPU is implemented. . • Misaligned memory access support for MSA: see Volume II, Appendix B “Misaligned Memory Accesses”. • Has2008 is required as of release 5 - Table 5.4, “FIR Register Descriptions”. • ABS2008 and NAN2008 fields of Table 5.7 “FCSR Register Field Descriptions” were optional in release 3 and could be R/W, but as of release 5 are required, read-only, and preset by hardware. • FPU $FCSR_{FS}$ Flush Subnormals / Flush to Zero behavior is made consistent with MSA behavior, in $MSACSR_{FS}$: Table 5.7, “FCSR Register Field Descriptions”, updated. New section 5.8.1.4 “Alternate Flush to Zero Underflow Handling”. • Volume I, Section 2.2 “Compliance ad Subsetting” noted that the L format is required in MIPS FPUs, to be consistent with Table 5.4 “FIR Register Field Definitions” . • Noted that UFR and UNFR can only be written with the value 0 from GPR[0]. See section 5.6.5 “User accessible FPU Register model control (UFR, CP1 Control Register 1)” and section 5.6.5 “User accessible Negated FPU Register model control (UNFR, CP1 Control Register 4)”
5.04	November 20, 2013	• No change to technical content.
6.00	March 31, 2014	• New notations x.bit[y], x.bits[y..z], x.byte[y], x.bytes[y..z], x.field, etc., added to section entitled Special Symbols in Pseudocode Notation, in Table Symbols Used in Instruction Operation Statements. • Combined tables entitled “Load and Store Operations Using Register + Offset Addressing Mode” and “FPU Load and Store Operations Using Register + Register Addressing Mode” into a single table; also includes PC-relative Addressing Mode. Updating Volume I for MIPS Revision 6: • Updated tables of instructions in Chapters 4 and 5, CPU and FPU Overview. All removed instructions and all new instructions are designated as such. • Reorganized discussion of Delay Slots and Forbidden Slots • MIPS Release 6 reserves SPECIAL2 opcode for use by MIPS. Previously it was available to customers for UDIs. • Statements about MIPS Release 6 not supported DSP and SmartMIPS removed.

Revision	Date	Description
6.01	August 20, 2014	Note: Edits are the union of all edits in AFP/ARM,MIPS32/64. • Edits are mainly cleanup from Revision 6.00, resolving consistency issues with Release 6. • Added FIR.FREP (new) Status.FR=0 emulation. • Added Table 6.1 FPU Register Models Availability and Compliance (new). Related lines throughout Vol I, modified to reflect removal of Status.FR=0 mode in R6 and availability of strictly 32-bit FPU in R6. • Table 1.2, Read/Write Register Field Notation: Added row for W0 • Chapter 5: Made tables consistent with R6 availability (removal/addition) of instructions. Some BC2* instructions are missing that will be added in next version.